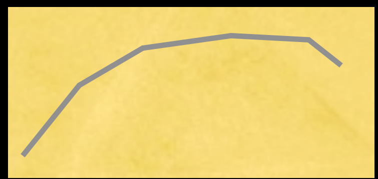
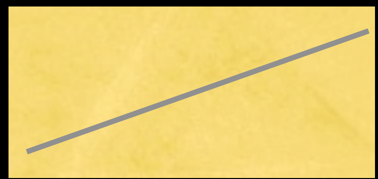
# Everything You Know (about Parallel Programming) Is Wrong! A Wild Screed about the Future

David Ungar
Sam Adams, Doug Kimelman, Mark Wegman
IBM Research

# How we got Smalltalk

- PARC living in the future with expensive but fast hardware + graphics
- cycles for
  - interpreter
  - dynamic dispatch
  - garbage collection
  - small methods
  - reusable collection classes

# Now, the future is manycore

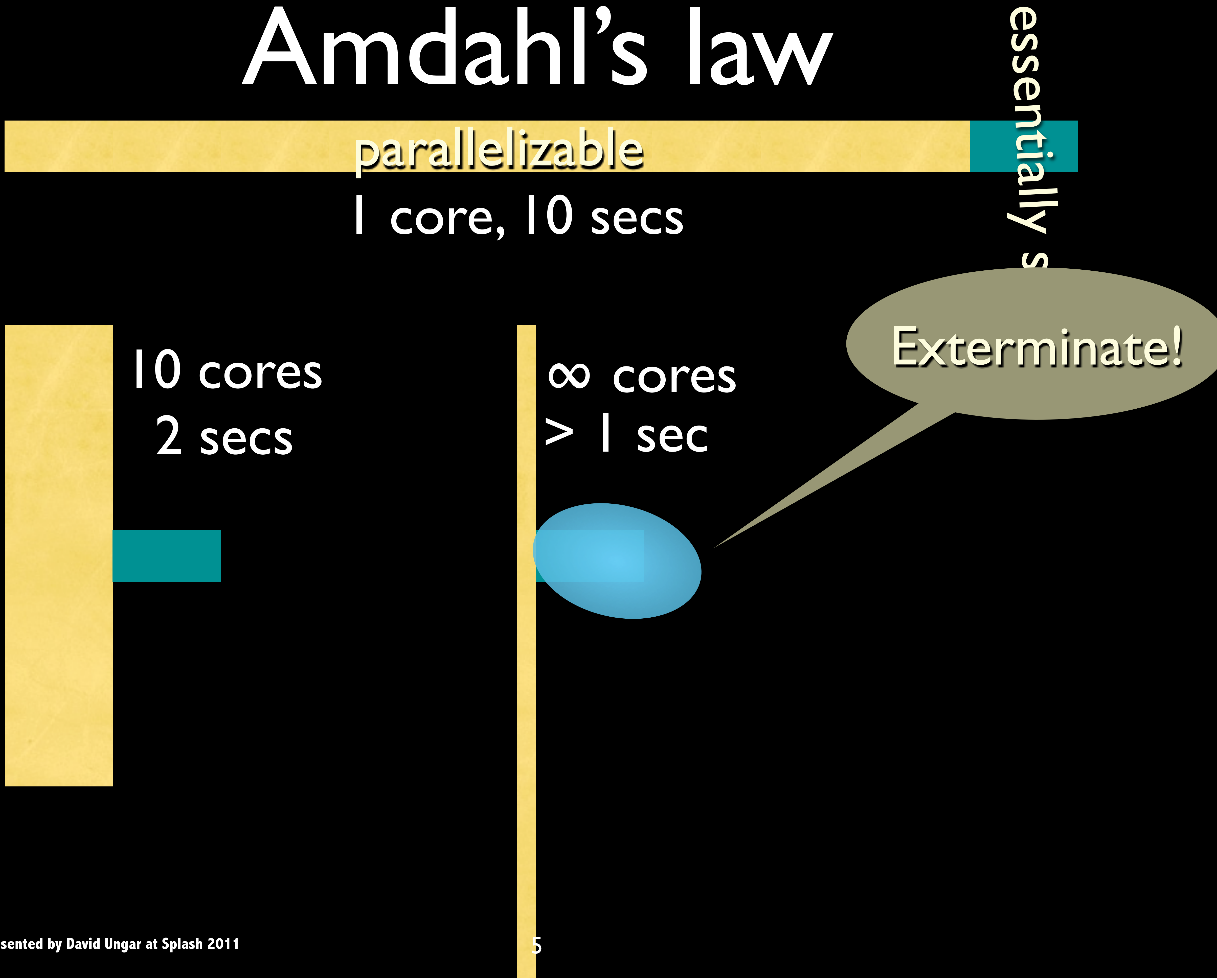
- Why?
  - Continued demand to handle more data
  - clock speed 
  - device density 
- What?
  - Much less (fast) memory per thread
  - Spatial locality critical for performance
  - Many (slower) cycles, all at the same time

# Fundamental Issues

performance

correctness

# Amdahl's law


parallelizable
essentially s
1 core, 10 secs
10 cores
2 secs
∞ cores
> 1 sec
Exterminate!

# Scaling

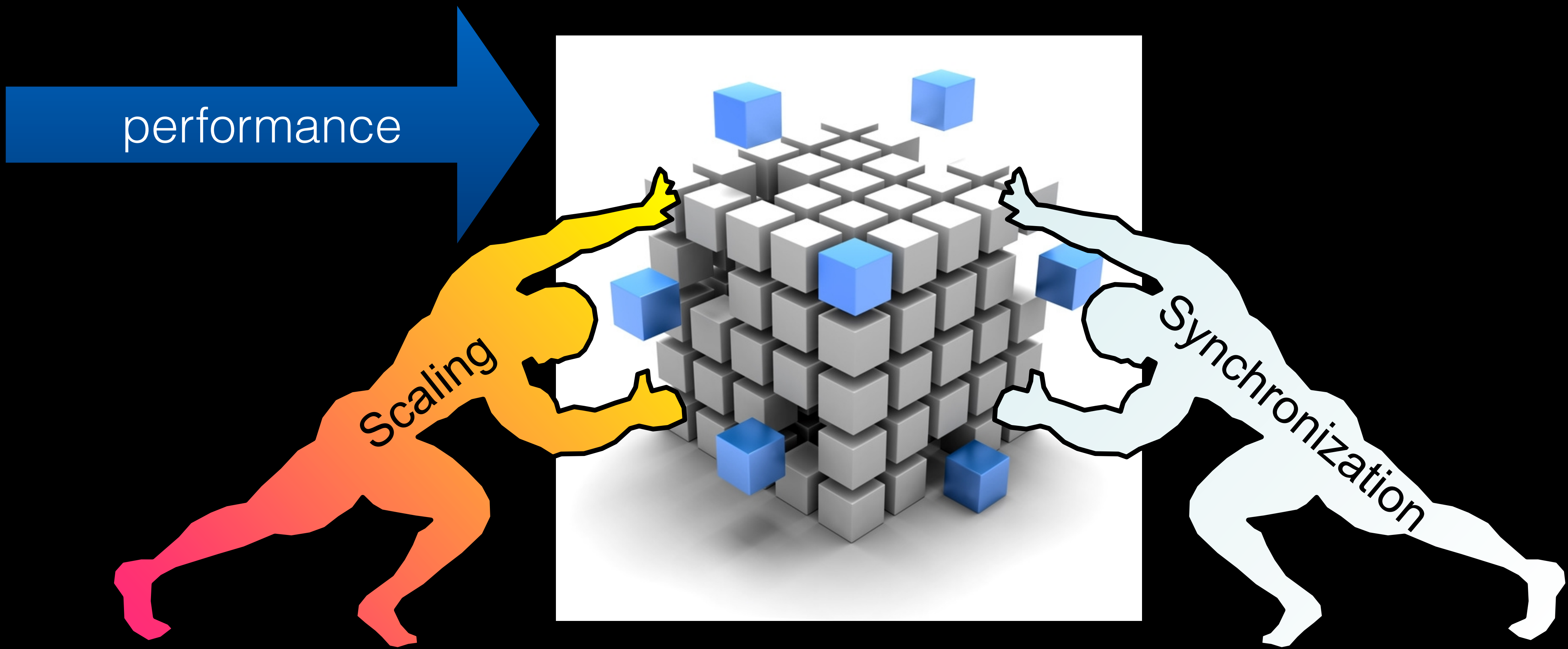
# implies

# ~~Serial~~ portion

# implies

# ~~synchronization~~

# Synchronization is Bad



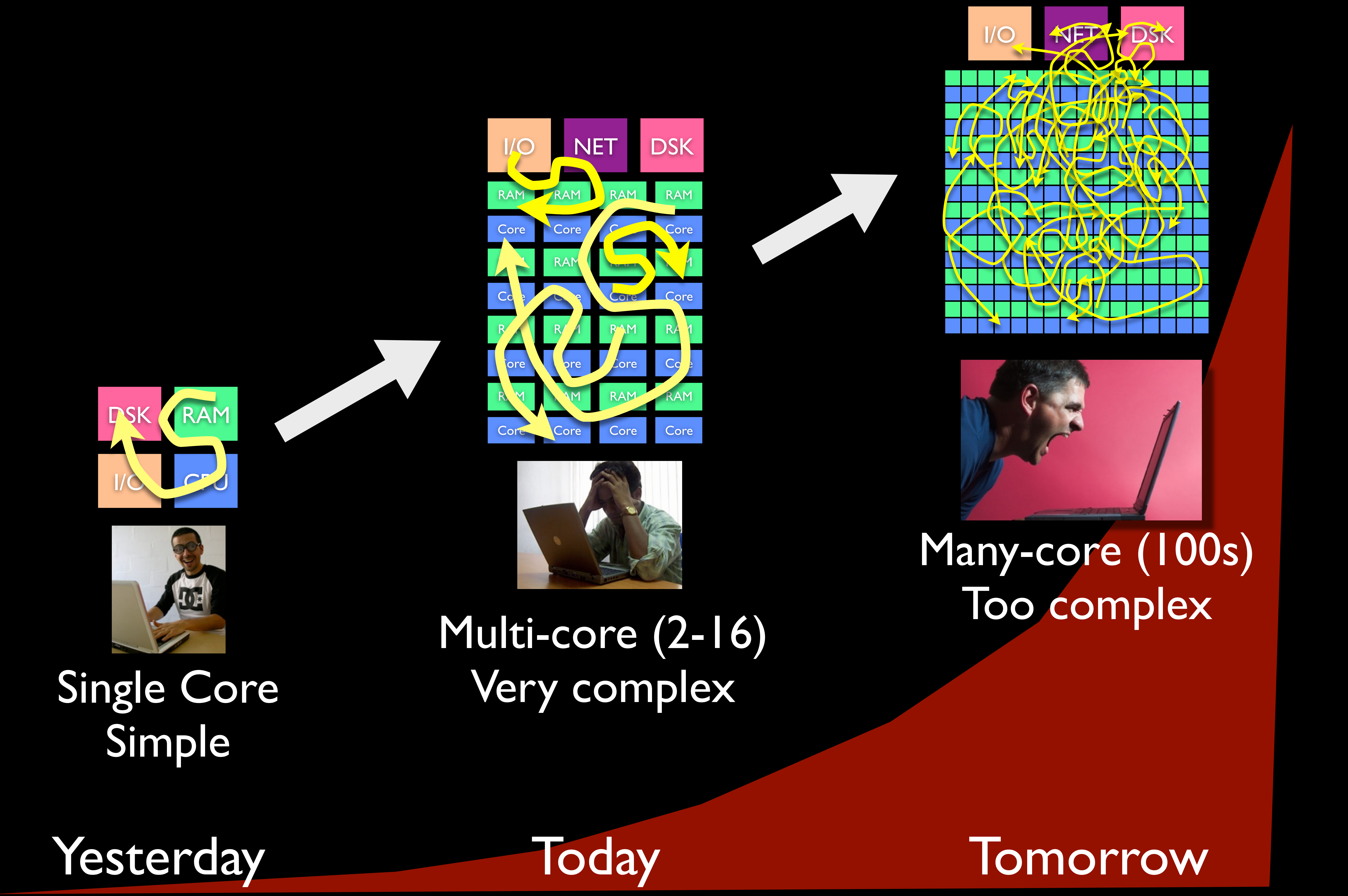

Too much → slow
Too little → errors

Why can't we eliminate synchronization
(in the programming paradigm)?

# Fundamental Issues

performance

correctness


DSK
RAM
I/O
CPU
Single Core
Simple
I/O
NET
DSK
RAM
Core
Multi-core (2-16)
Very complex
I/O
NET
DSK
Many-core (100s)
Too complex
Yesterday
Today
Tomorrow

Too hard to
get it right
when parallel

Cannot even try to
get it right
without synchronization





10

# The future: No sync at all

- "anti-lock"
- "race-and-repair"
- "end-to-end nondeterminism"
- Without synchronization:
  - will not always get exact answers

performance

correctness

Get it wrong,
quickly,
but still
right enough

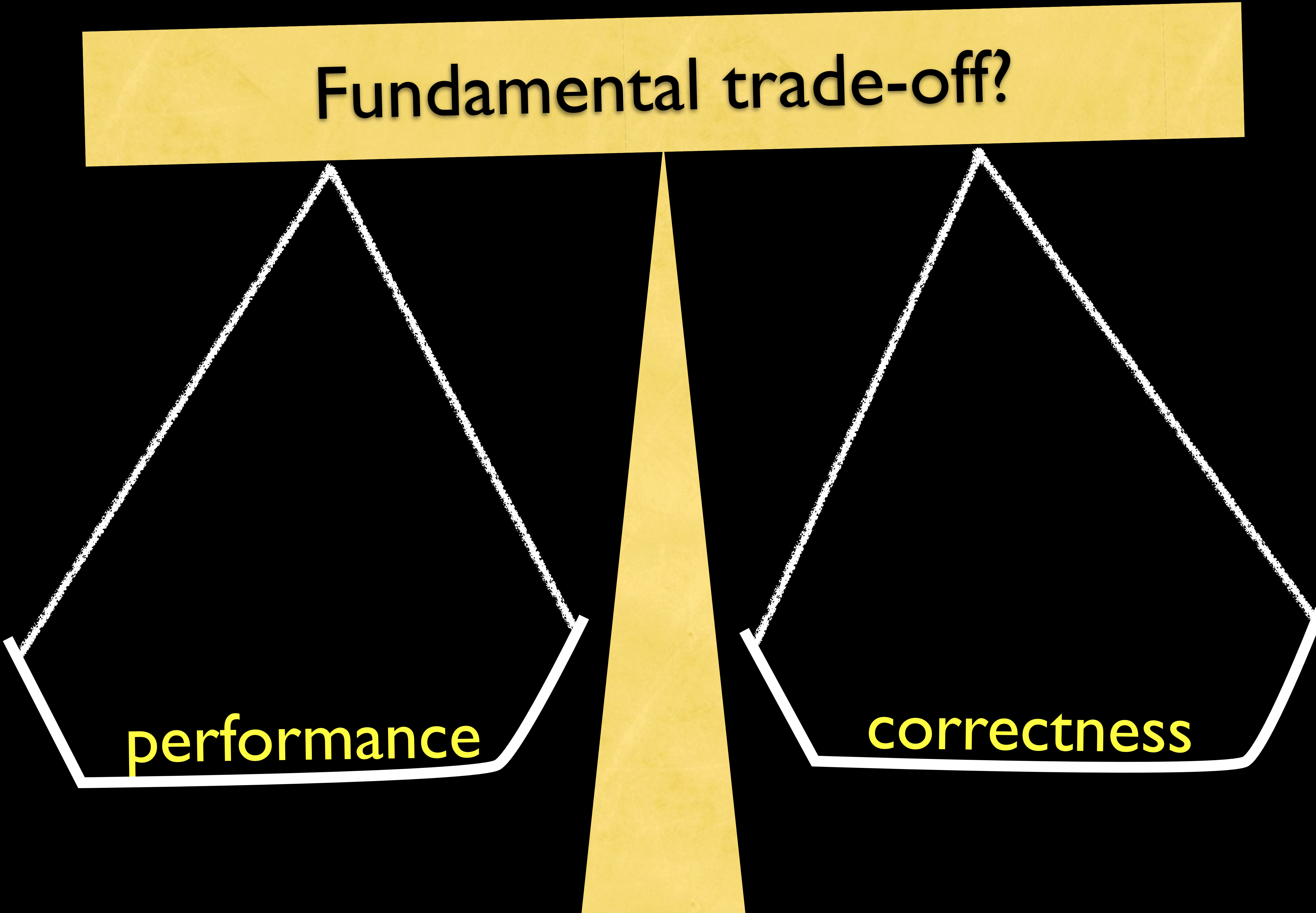




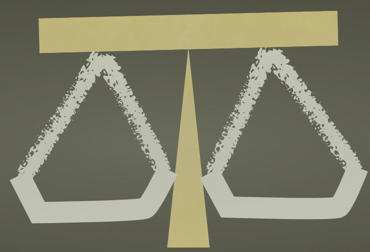


12

# Fundamental trade-off?

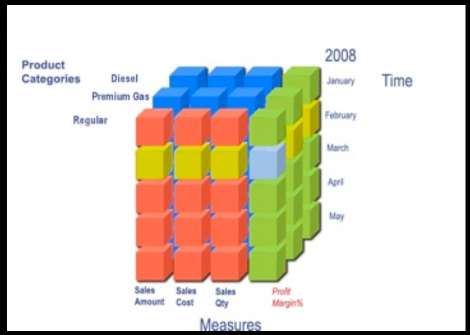


# Fundamental



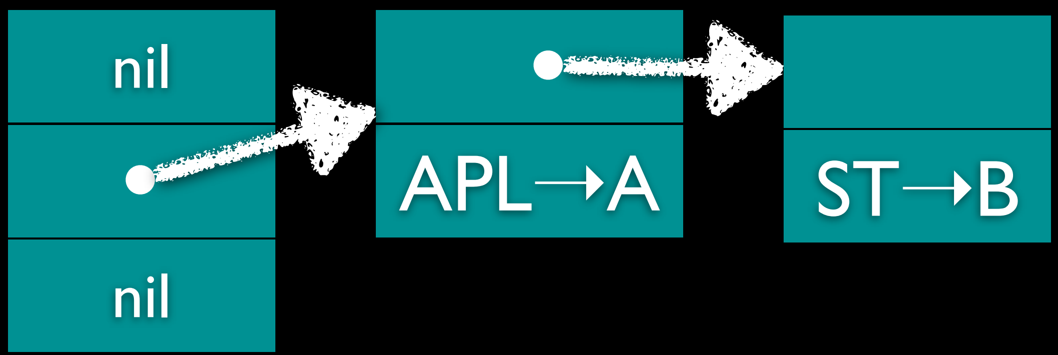
Ensembles & Adverbs

Fresheners & Breadcrumbs



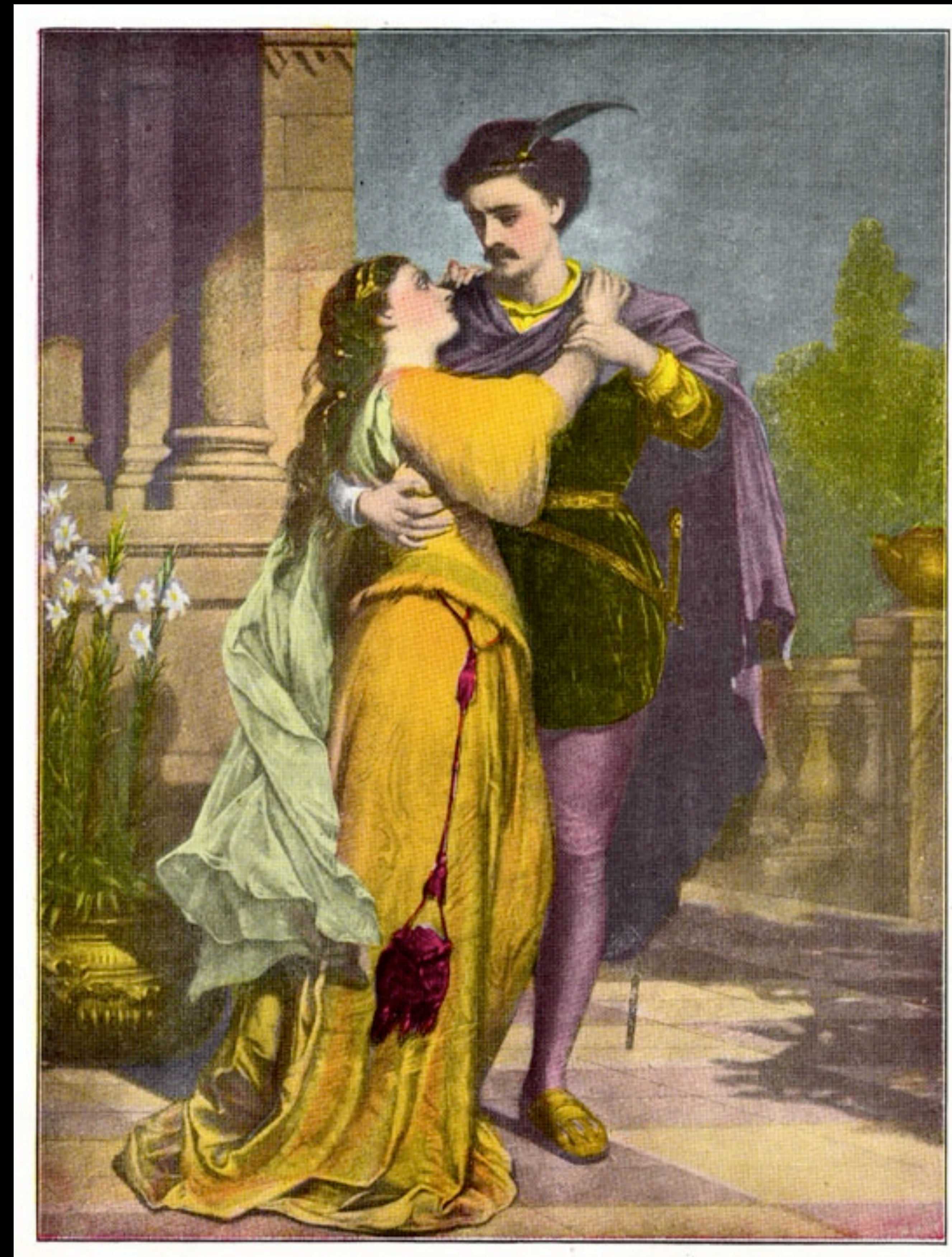

Mitigate, Race, Repair

Locals & Breadcrumbs

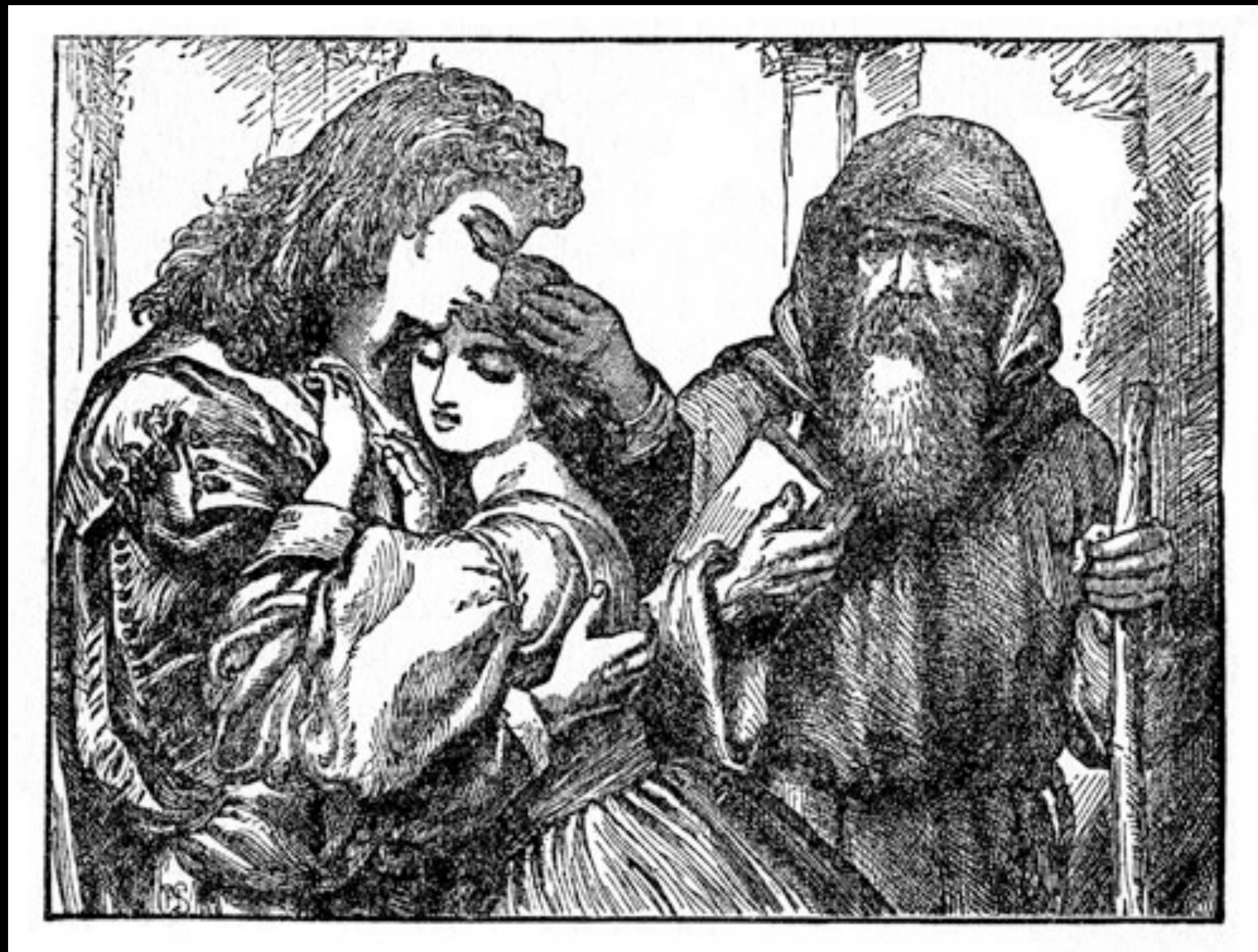


# Romeo and Juliet

Spoiler alert!



http://karenswhimsy.com/romeo-and-juliet.shtm



# Friar Lawrence hatches a plan







16

Juliet fakes death with a drug.

Friar John is sent to tell Romeo

- John is delayed by quarantine
- Servant tells Romeo that Juliet is dead
- Romeo goes to tomb
- Romeo kills himself
- Friar Lawrence arrives with message to Romeo

Juliet wakes

to find a

dead Romeo





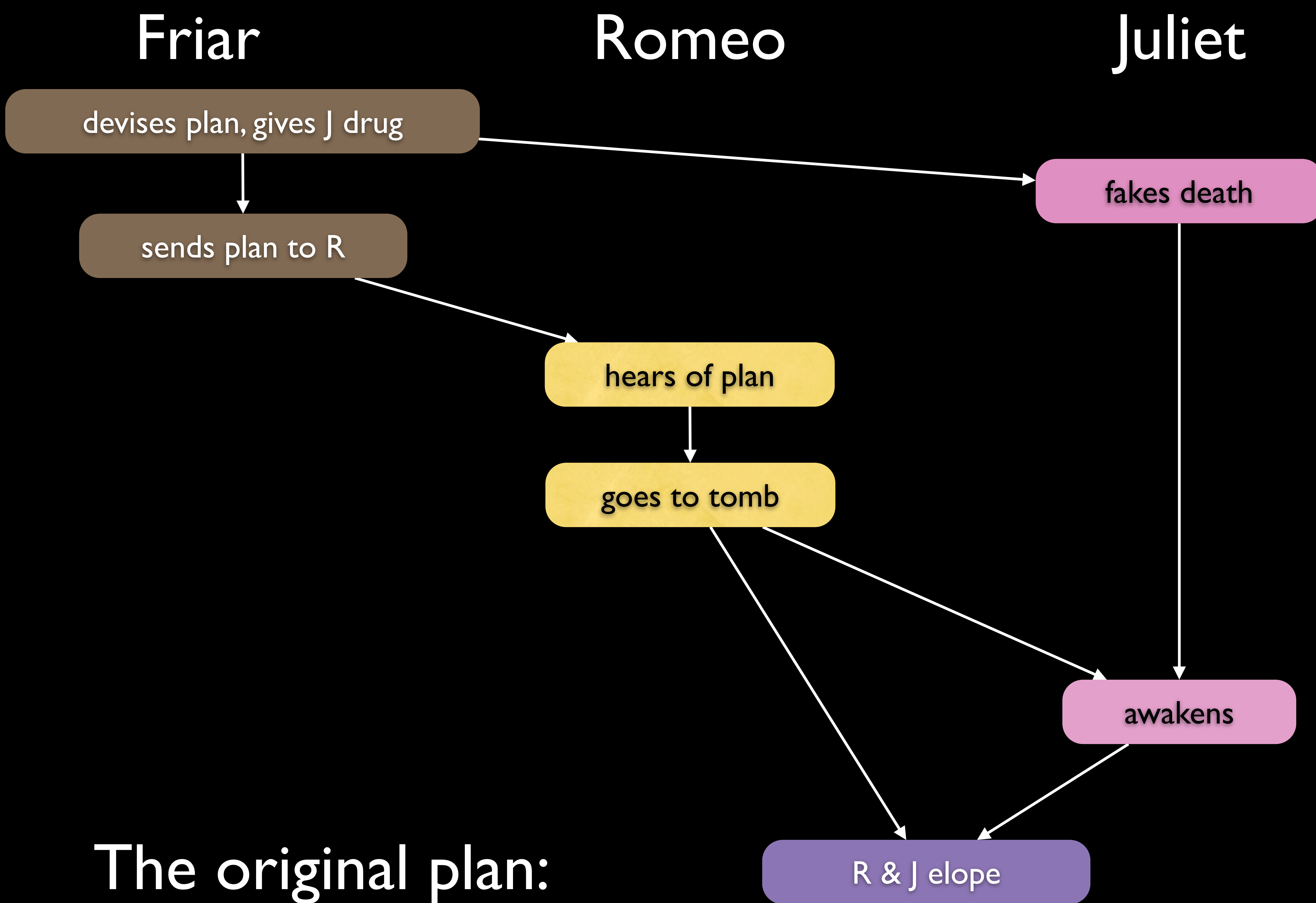


19

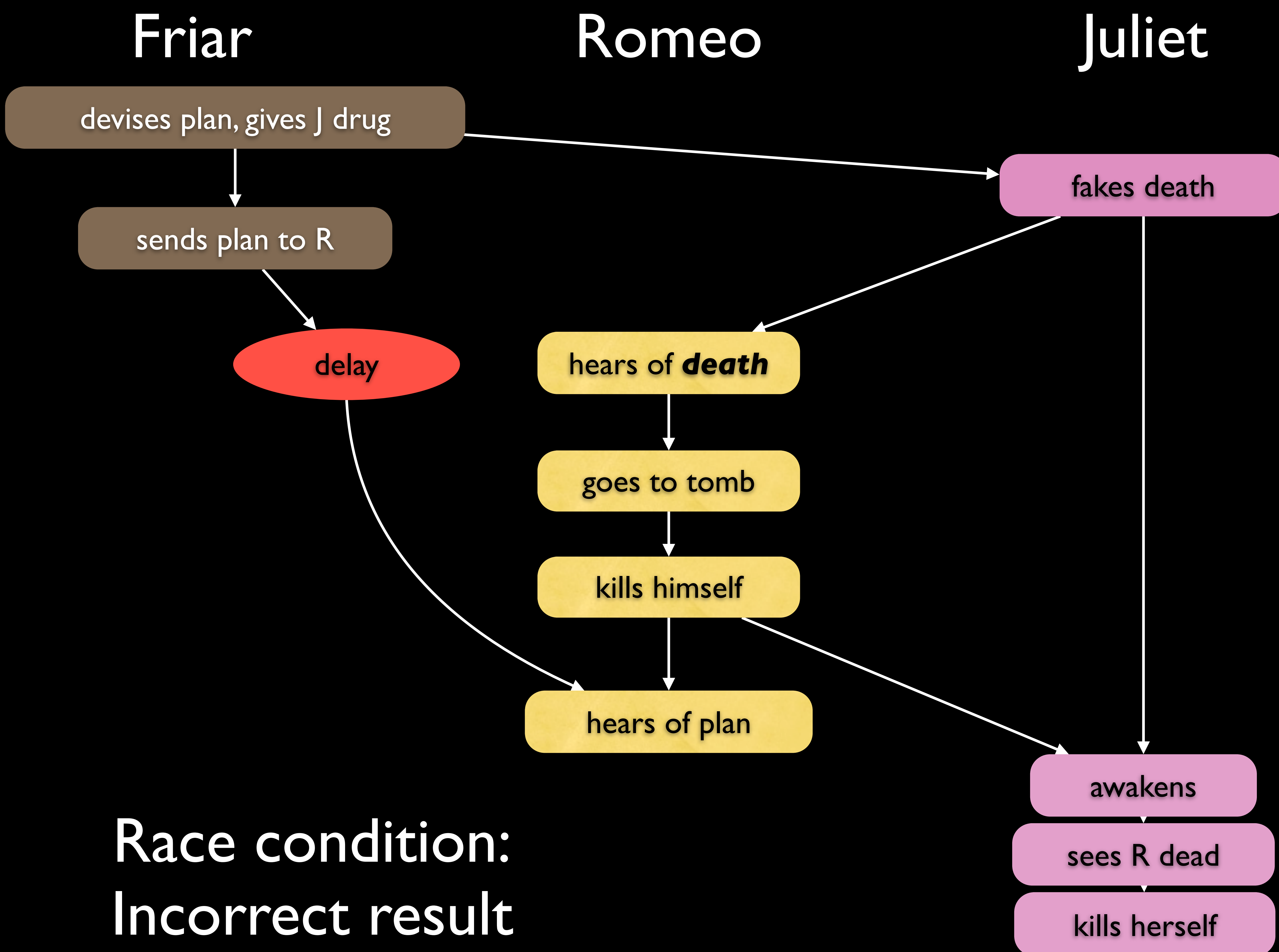
# Juliet kills herself

# Summary

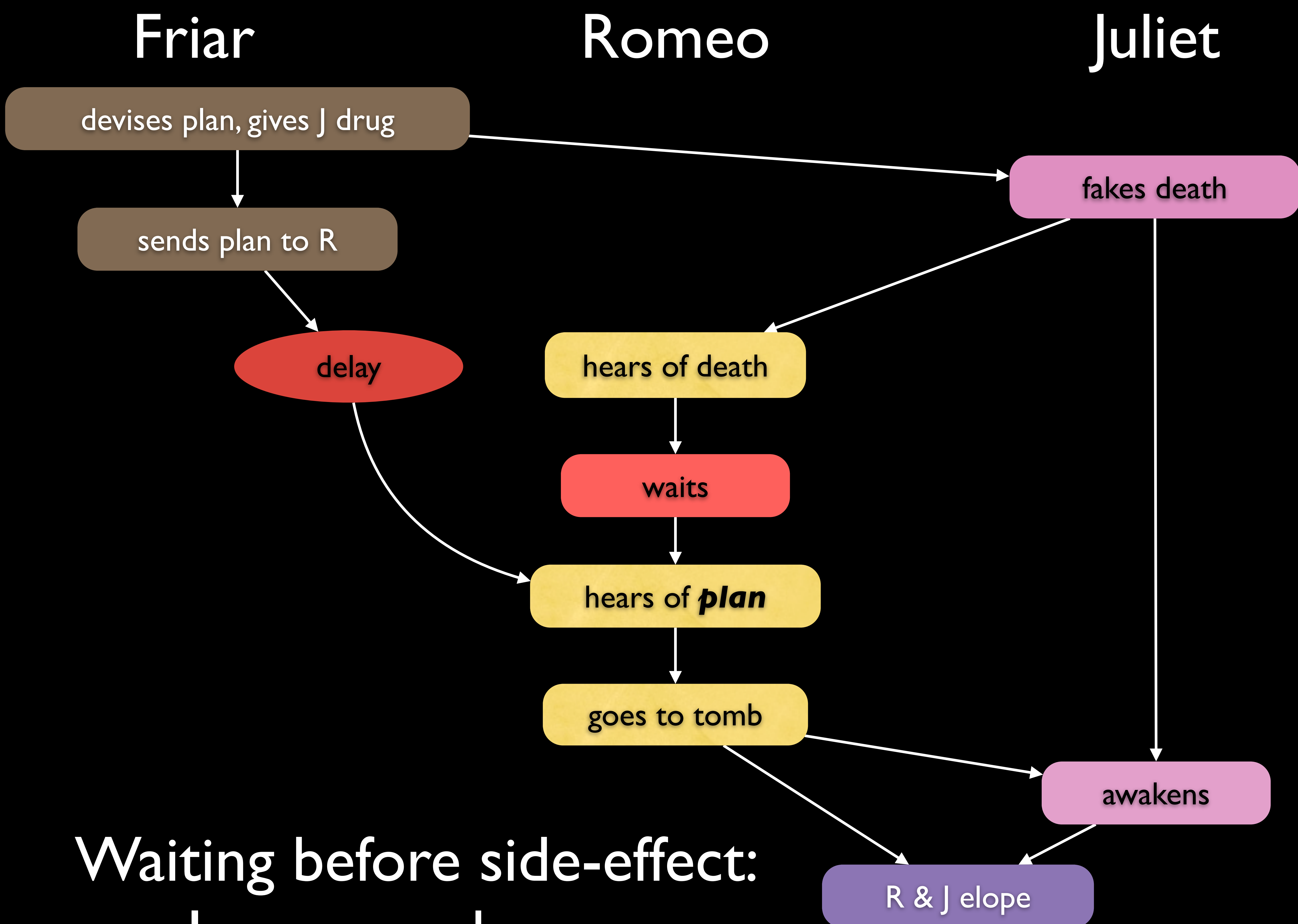
- Juliet feigns death to avoid marrying Paris
- Friar Lawrence sends Friar John to tell Romeo of plan
- John is delayed by quarantine
- Servant tells Romeo that Juliet is dead
- Romeo goes to tomb
- Romeo kills himself
- Friar Lawrence arrives with message to Romeo
- Juliet wakes, sees Romeo dead, kills herself



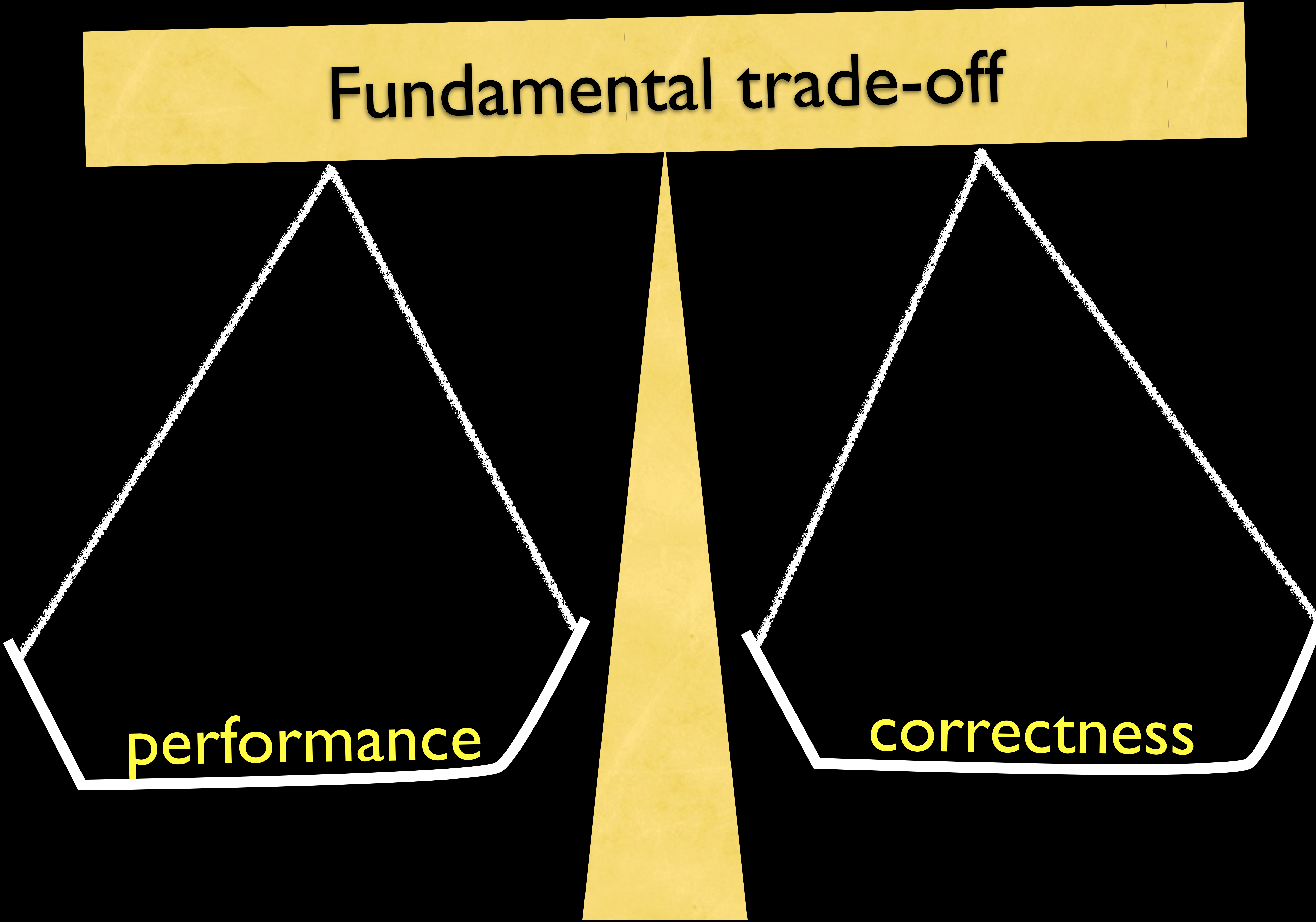

The original plan:
A happy ending




# Race condition: Incorrect result




# Waiting before side-effect: Improves chances


Fundamental trade-off
performance
correctness

# Other Ideas
# (not really covered)





26

# “Lock-Free” algorithms

- Critical section limited to atomic instructions
  - compare-and-swap
  - lwarx & stwx
- Instruction may “fail” forcing a retry loop
- No waiting visible to programmer
- But atomic instructions implicitly synchronize

➡ may not scale!

# Read-Copy Update

- Readers run concurrently with updaters
- Updaters update a copy if needed
- After all readers done, updaters serially swap-in updated copy
- Handles removal

- Good lessons to learn
- Still pays synchronization costs, esp. for updating: guaranteed to not miss updates

# Functional Programming

- Lack of side-effects hides many ordering dependencies
- But, a poor match for modeling stateful systems
- Functional composition: f(g(x))
  - still induces ordering dependencies

➡ some synchronization required

# Other Deterministic Programming Approaches

- Let the programmer specify dependencies
- System reorders and parallelizes execution
- Does not push programmer hard enough to relinquish determinism

# Actors

- Determinism within an actor eases programming task

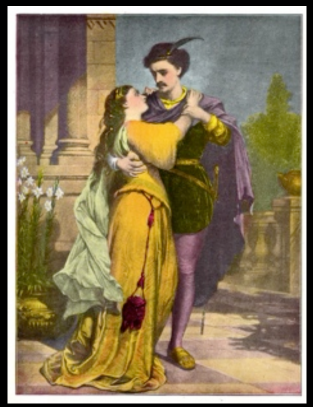
- But, message arrival ordering still creates need to deal with nondeterminism

Other approaches still cling to correctness

# Root cause: Our Attraction to Certainty

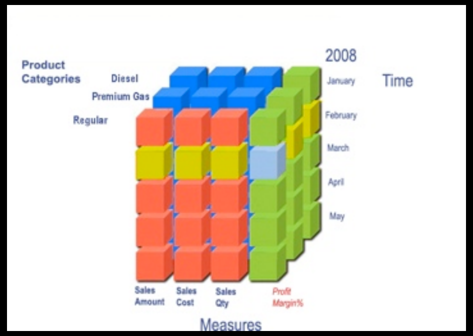
- Definite state
  - x holds 17
- Definite order
  - input → process → output
  - serialized message queues
- Definite results
  - 1 + 1 = 2

Fundamental



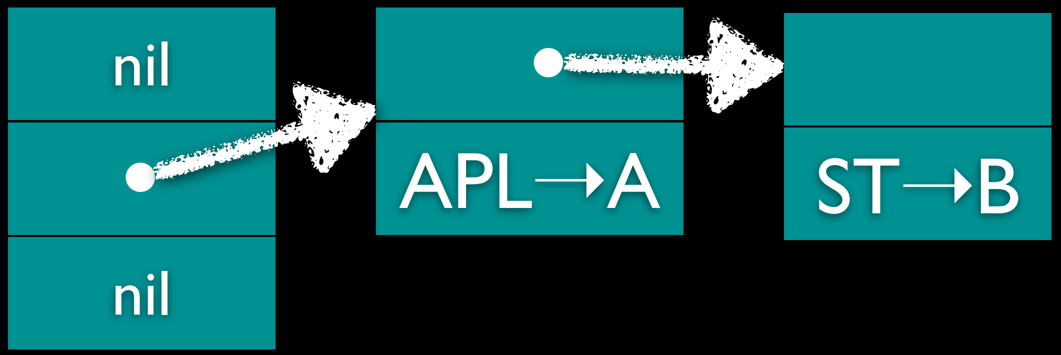
Ensembles &
Adverbs

Fresheners &
Breadcrumbs




Mitigate,
Race,
Repair

Locals &
Breadcrumbs



# Biology, not Math

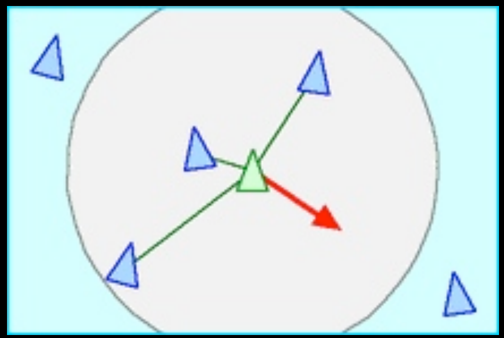
Massive parallelism with state:

Many locally (re)acting individuals

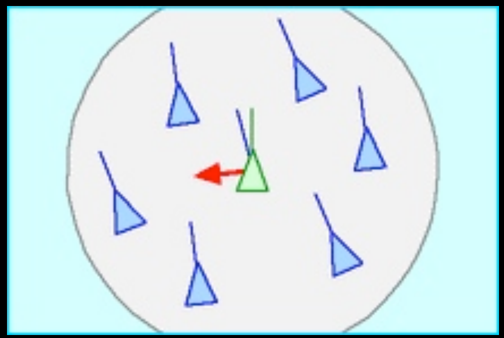
Surprisingly complex overall behavior

**Emergence**

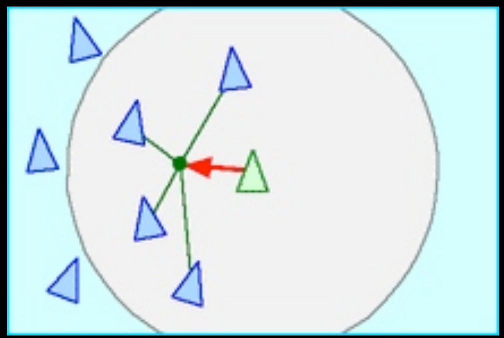
# Birds don't need π calculus



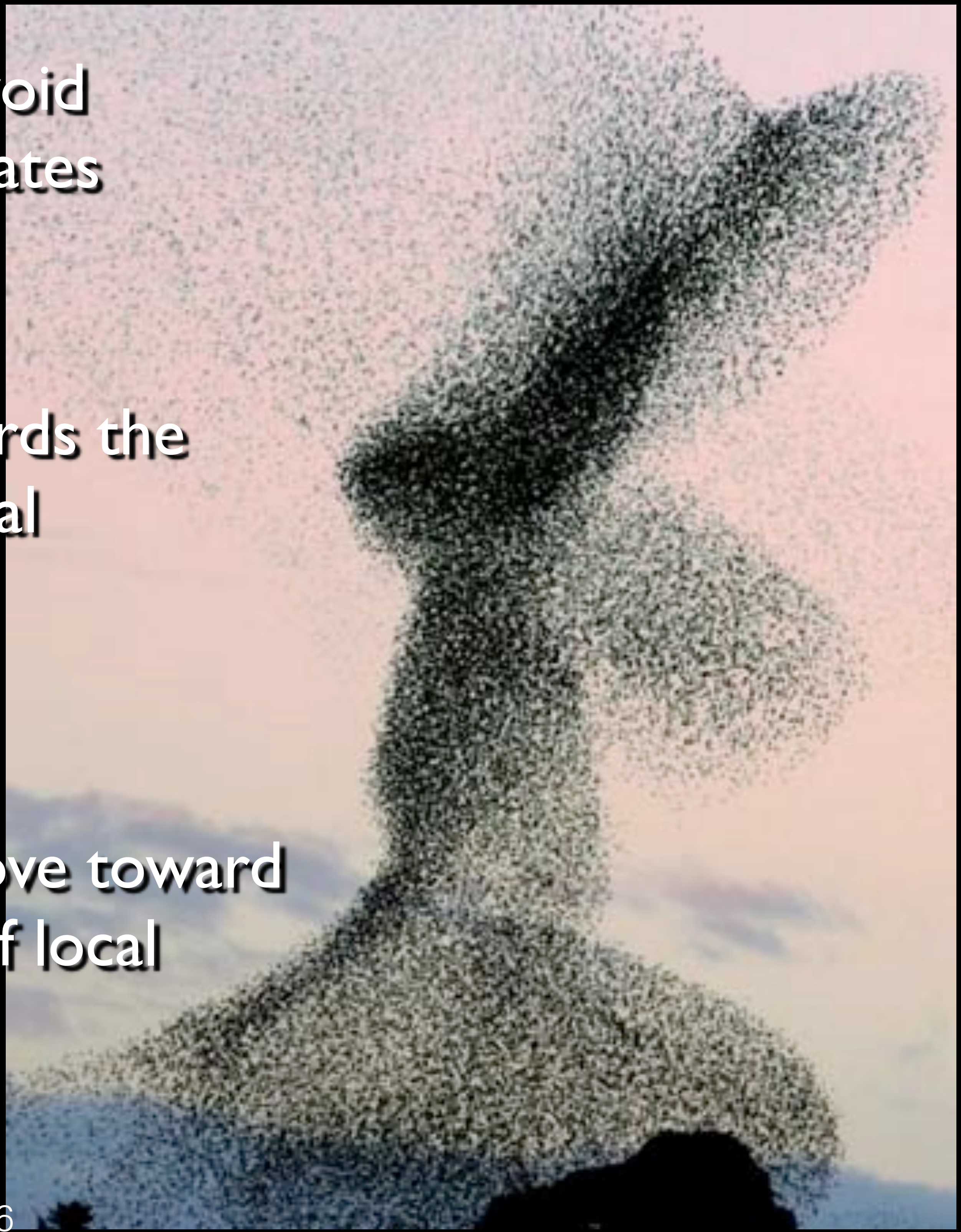
**Separation:** steer to avoid crowding local flockmates



**Alignment:** steer towards the average heading of local flockmates

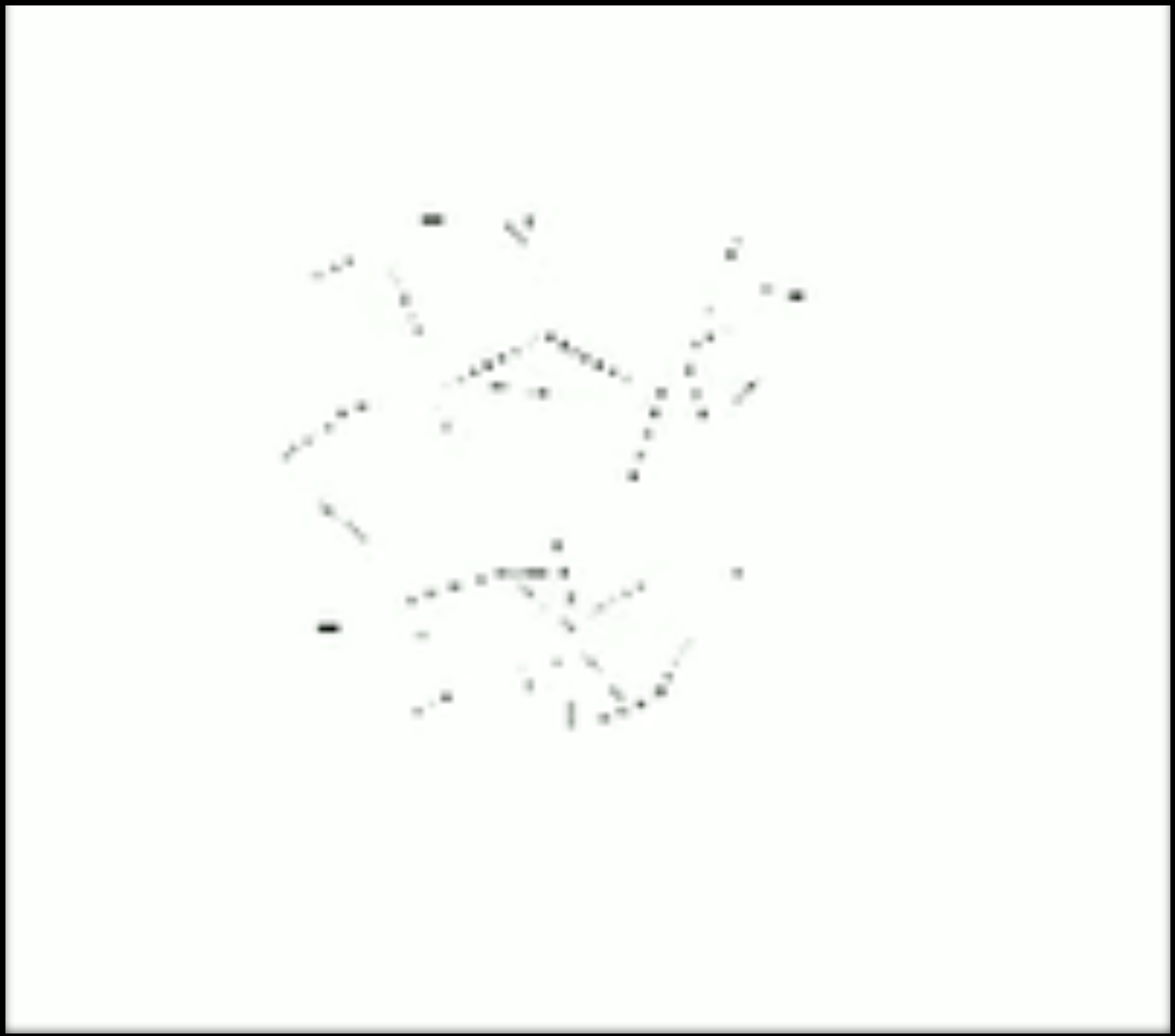


**Cohesion:** steer to move toward the average position of local flockmate



Craig Reynods, 1986, Boids
(Flocks, Herds, and Schools: a Distributed Behavioral Model)

# 50 SlyBoids, 50 Tilera cores



# Ensemble

One & Many

Parallel activities

Unsynchronized







38

# Ensemble computation varies from

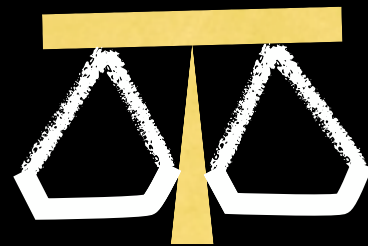
Independent & Parallel ⟷ execution strategies ⟷ Dependent & Serial

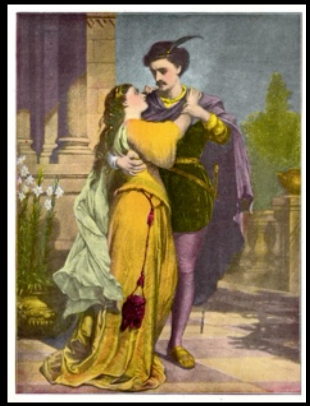
Ensemble of results ⟷ result handling strategies ⟷ Reduced to a Single Object

## Idea:

Separate how from what (and who);
factor out the strategy:
subject + verb + adverb
receiver.selector(argument --modifier)

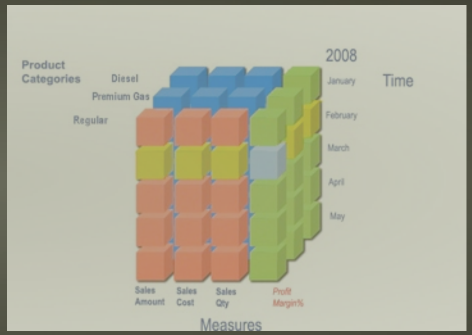
Fundamental





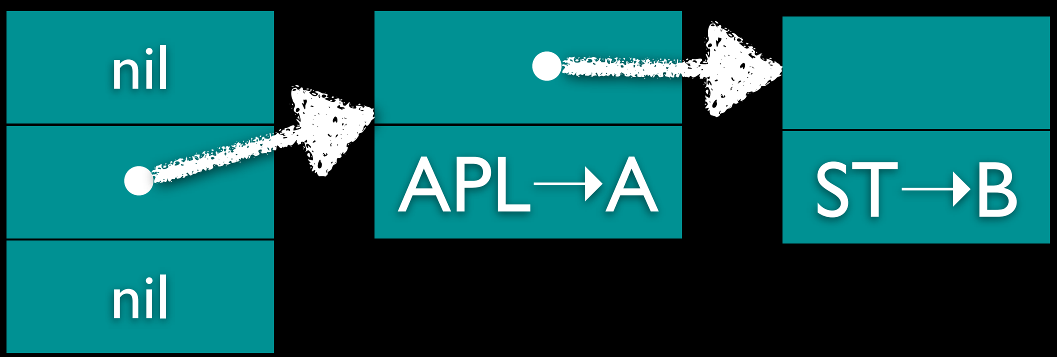
Ensembles & Adverbs

Fresheners & Breadcrumbs




Mitigate, Race, Repair

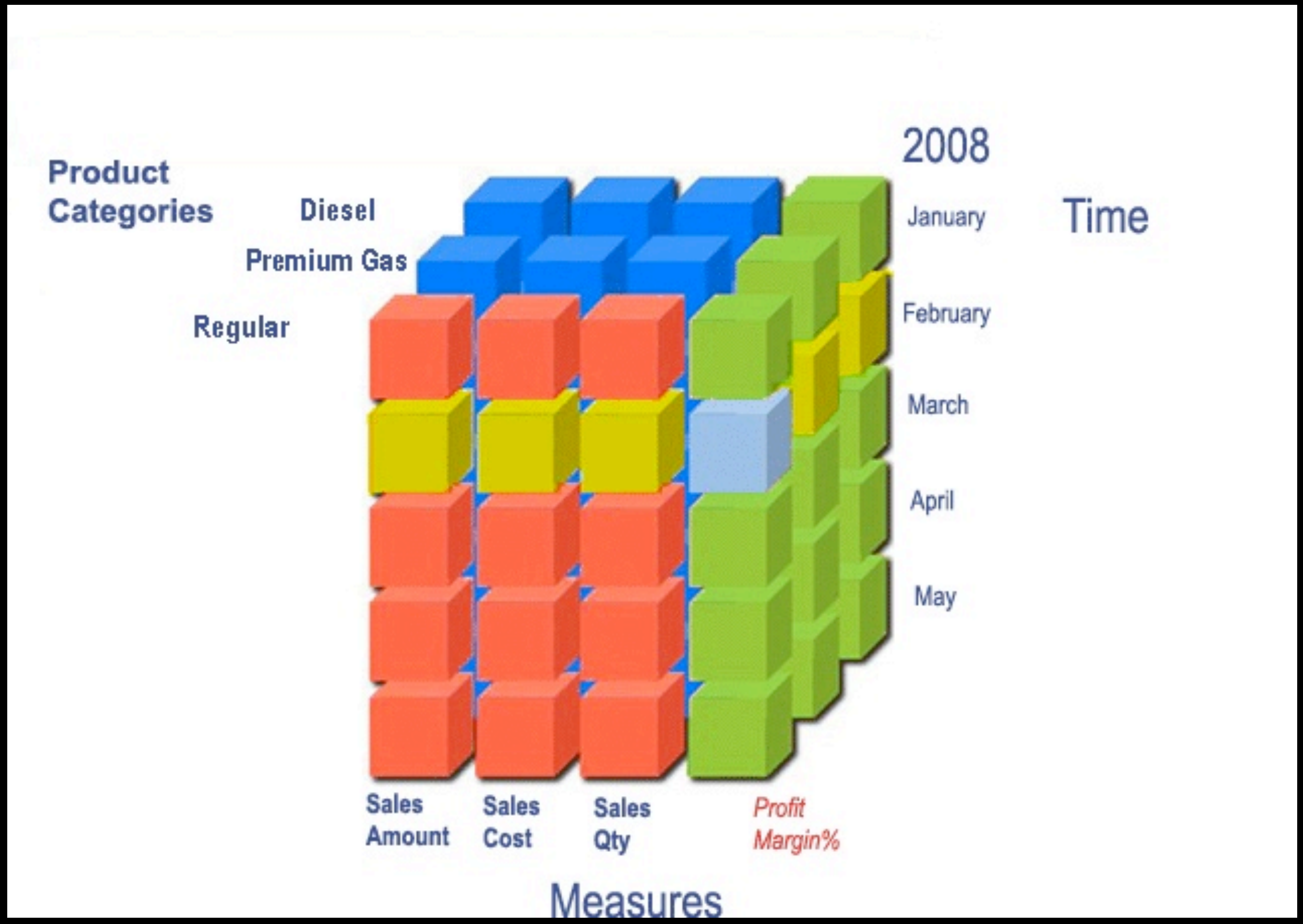
Locals & Breadcrumbs



# What's a cube?

(OLAP = Online Analytical Processing)

- To a first approximation: It's a multidimensional spreadsheet



# Our OLAP Cubes' Features

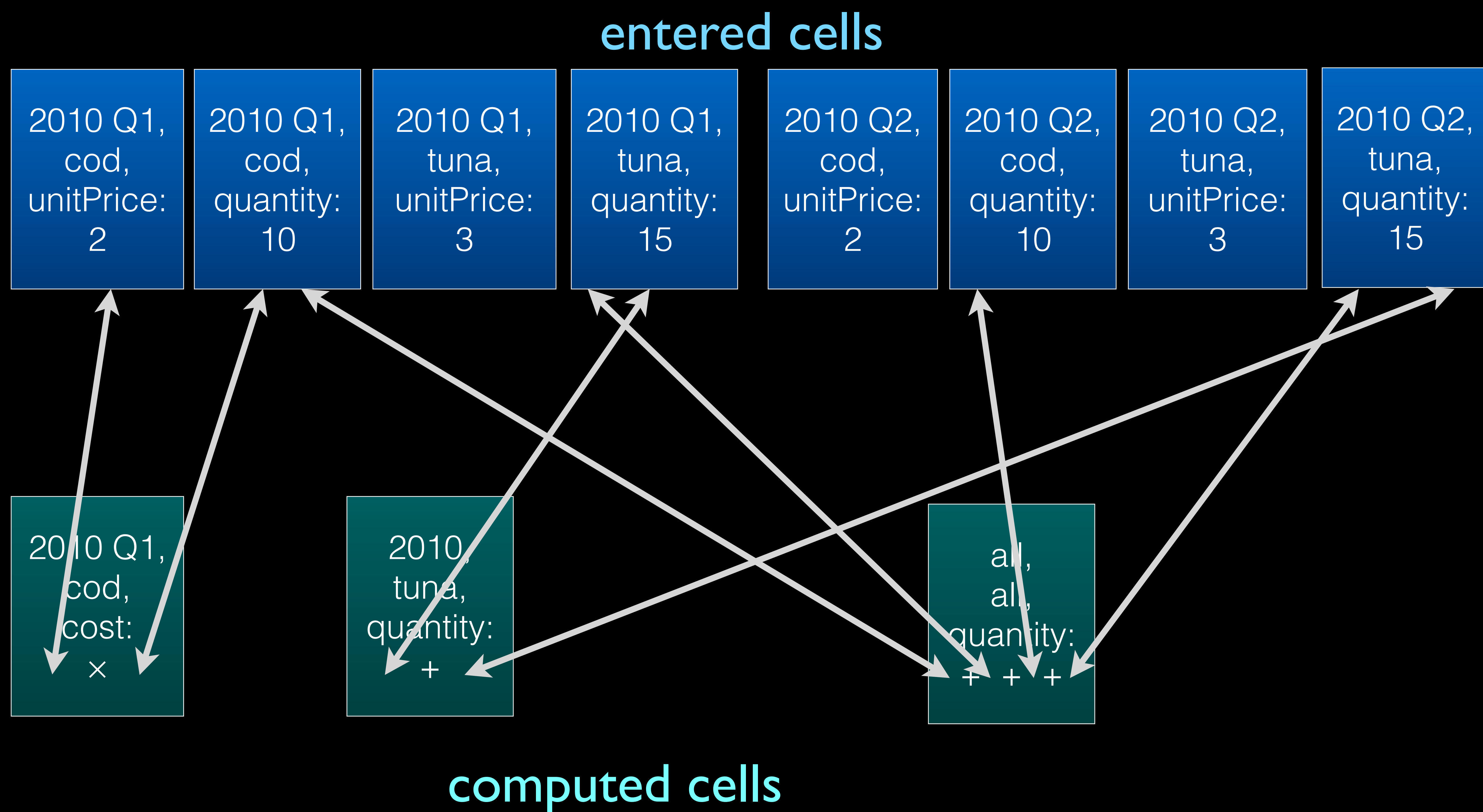
- In-memory – to be practical for interactive update / recalculate
- Not represented by a standard Relational Database, thus **M**OLAP
- Write-back – users update values e.g. for financial forecasting / budgeting
- Concurrent – up to 100's or 1000's of users

# Users Want Scalability

- Budget deadlines, 1000's of users, some doing vast queries, many others doing detailed entry and review
- Scaling / Performance wall (long running reads, serializing writes)
  - Readers-writer lock contention

# Data cells linked by one-way constraints

- Could be any (acyclic) shape
- “Entered Cells” = user types in data
- “Computed Cells” = hold sums, etc.
  - Aggregates & Formulae results
  - Computed on demand
  - Cache results for performance


entered cells
2010 Q1, cod, unitPrice: 2
2010 Q1, cod, quantity: 10
2010 Q1, tuna, unitPrice: 3
2010 Q1, tuna, quantity: 15
2010 Q2, cod, unitPrice: 2
2010 Q2, cod, quantity: 10
2010 Q2, tuna, unitPrice: 3
2010 Q2, tuna, quantity: 15
2010 Q1, cod, cost: ×
2010, tuna, quantity: +
all, all, quantity: + + +
computed cells

# Naive Caching

0A. unitPrice = 2, quantity = 10

1A. Alice requests cost

2A. Alice sees empty cache

3A. value calculated is 20

4A. 20 is cached to save recalculation

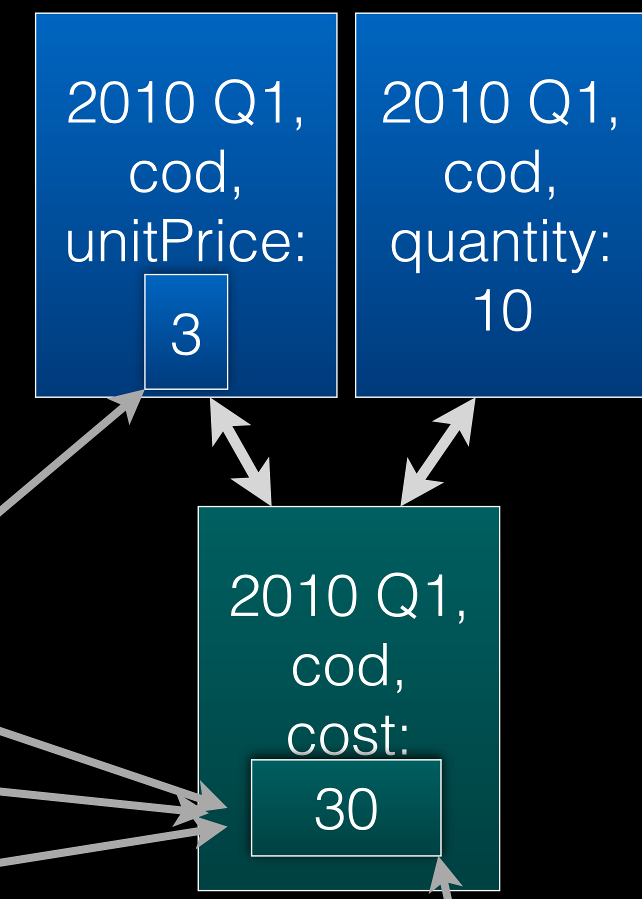
1B. Bob changes unitPrice, invalidates cache

1C. Cathy requests cost

2C. Cathy sees empty cache

3C. Cost recalculated & cached

4C. Cathy gets fresh cost

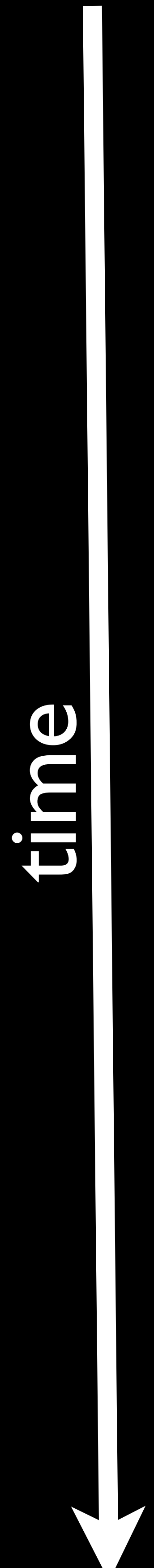



time

Works when serial

Fails when concurrent

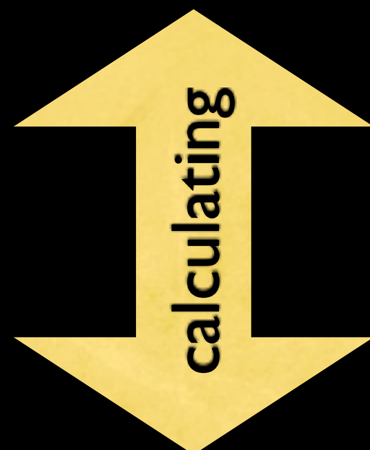
# Naive caching fails: leaves stale result cached forever




0A. unitPrice = 2

1A. Alice requests cost value

2A. calculation commences




1B. Bob changes unitPrice to 10

2B. cost cache is invalidated

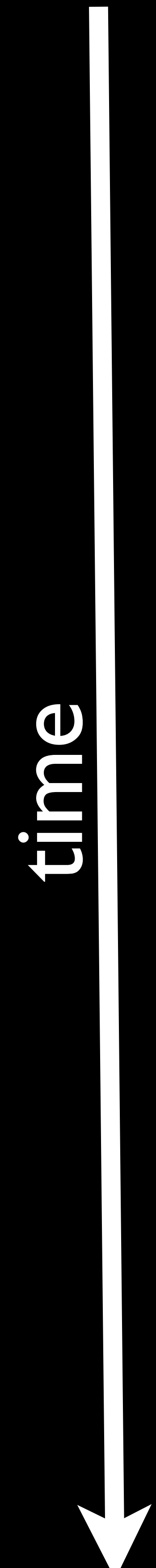
3A. calculation finishes,
stores **wrong** value in cache

1C. Cathy requests cost,
reads **wrong** value from cache

1D. Dan requests cost,
reads **wrong** value from cache

1E. Elly May requests cost,
reads **wrong** value from cache

# Naive parallel solution:
# lock allows N readers OR one writer




0A. unitPrice = 2

1A. Alice requests cost value, gains lock

2A. calculation commences


3A. calculation finishes,
stores **iffy** value in cache
releases lock

1B. Bob tries to change unitPrice,
has to wait for lock


2B. Bob gets lock,
changes unitPrice,
invalidates cost cache,
releases lock

1C. Cathy requests cost,
gets lock, sees empty cache,
recalculates & caches,
reads **right** value from cache
releases lock

# Asynchronous freshener eventually fixes error without locking

time

0A. unitPrice = 2

1A. Alice requests cost value

2A. calculation commences


1B. Bob changes unitPrice to 10

2B. cost cache is invalidated

3A. calculation finishes,
stores **wrong** value in cache

"race and repair"

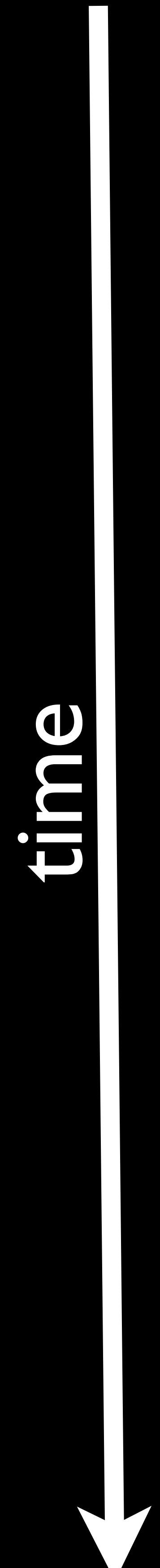
1C. Cathy requests cost,
reads **wrong** value from cache

*1F. Freshener recalculates cost cell, caching result*

1D. Dan requests cost,
reads **right** value from cache

1E. Elly May requests cost,
reads **right** value from cache

# Breadcrumbs: Avoid caching (some) stale results
# ***Mitigate*** nondeterminism




1A. Alice requests cost value

2A. Alice drops her breadcrumb

3A. calculation commences


1B. Bob changes unitPrice to 10

2B. cost cache is invalidated

3B. Bob drops his breadcrumb

4A. calculation finishes

5A. Alice picks up Bob's breadcrumb,
aborts cache store, gets reasonable result

1C. Charles requests cost,
cache is empty, recalculates and caches **right** result

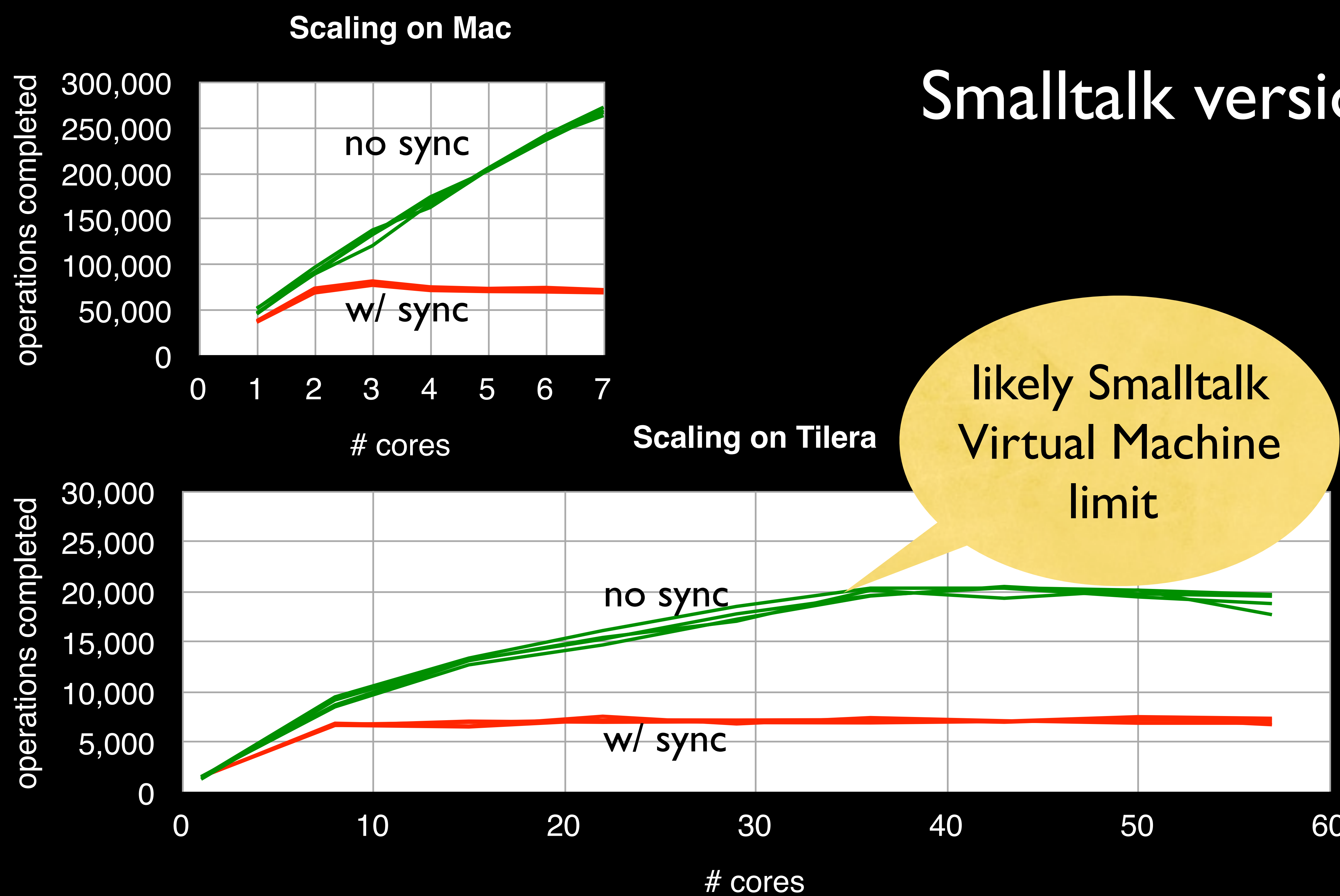
1D. Doris requests cost,
reads **right** value from cache

1E. Ephraim requests cost,
reads **right** value from cache

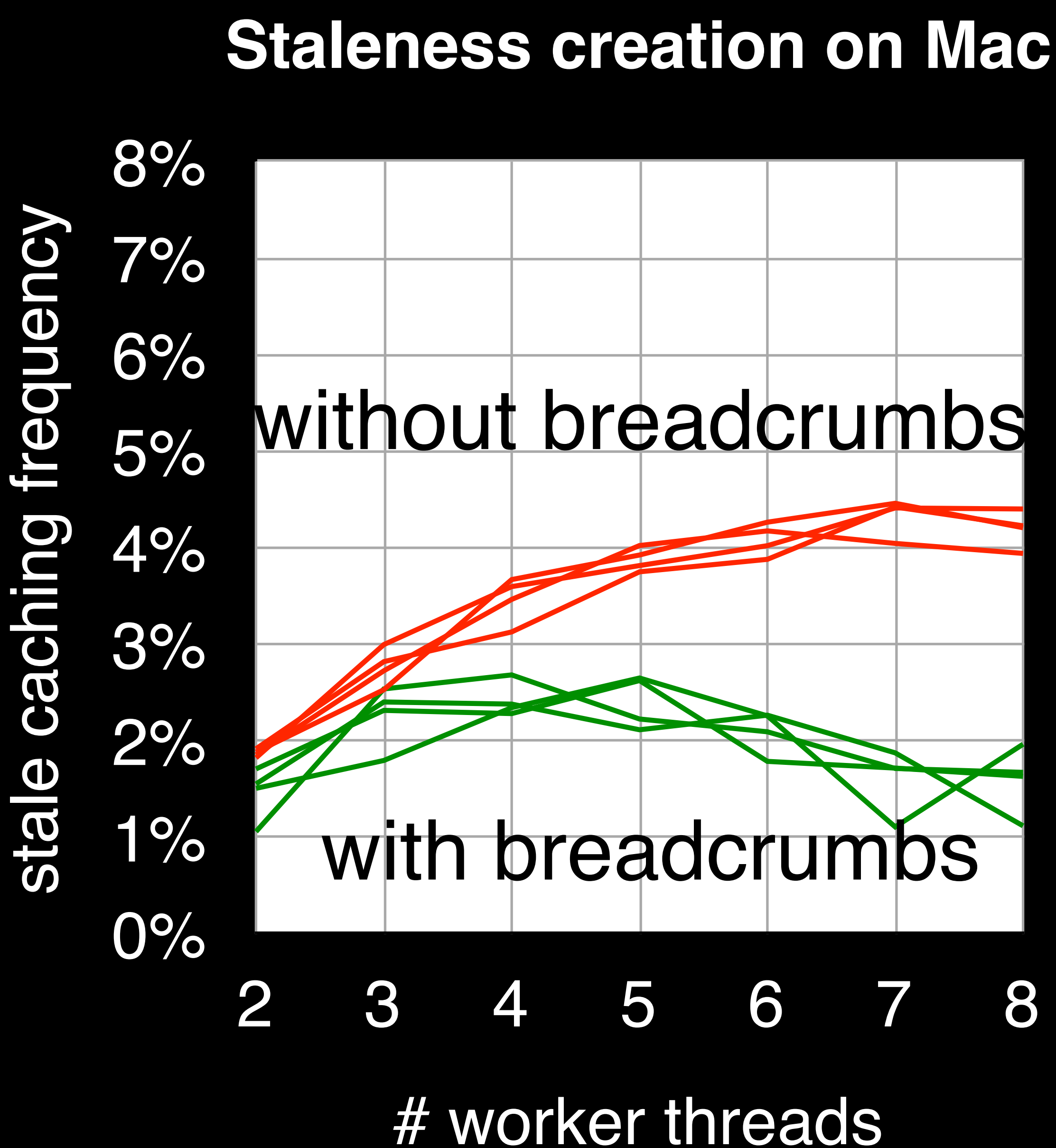
## Imperfect
## Many variations

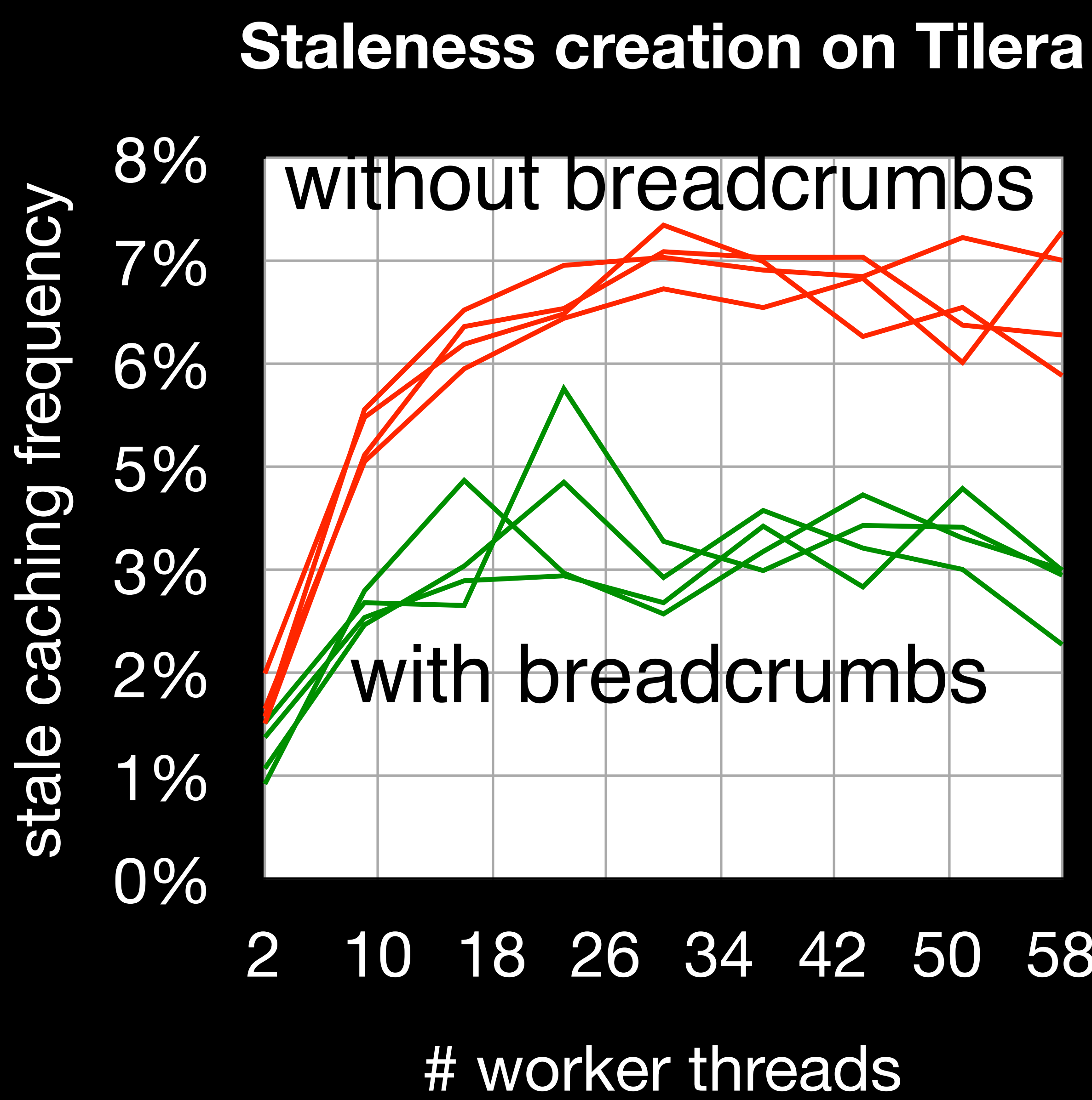
# Synchronization prevents scaling

Smalltalk version



# Reducing incidence of staleness without sync



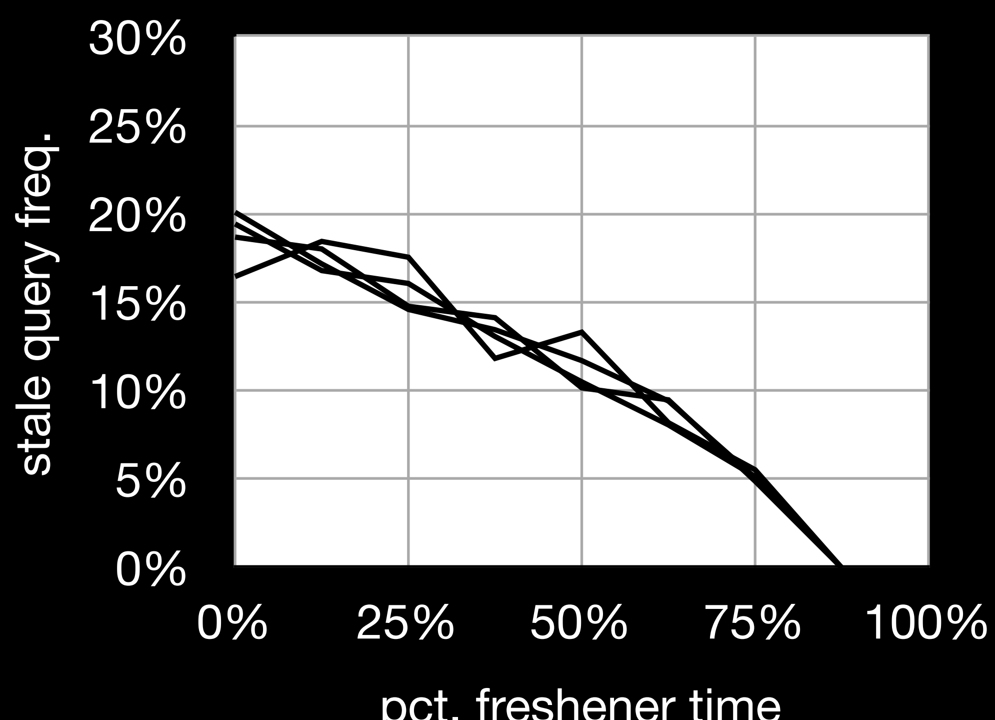




600 cell Fish Market, Smalltalk model

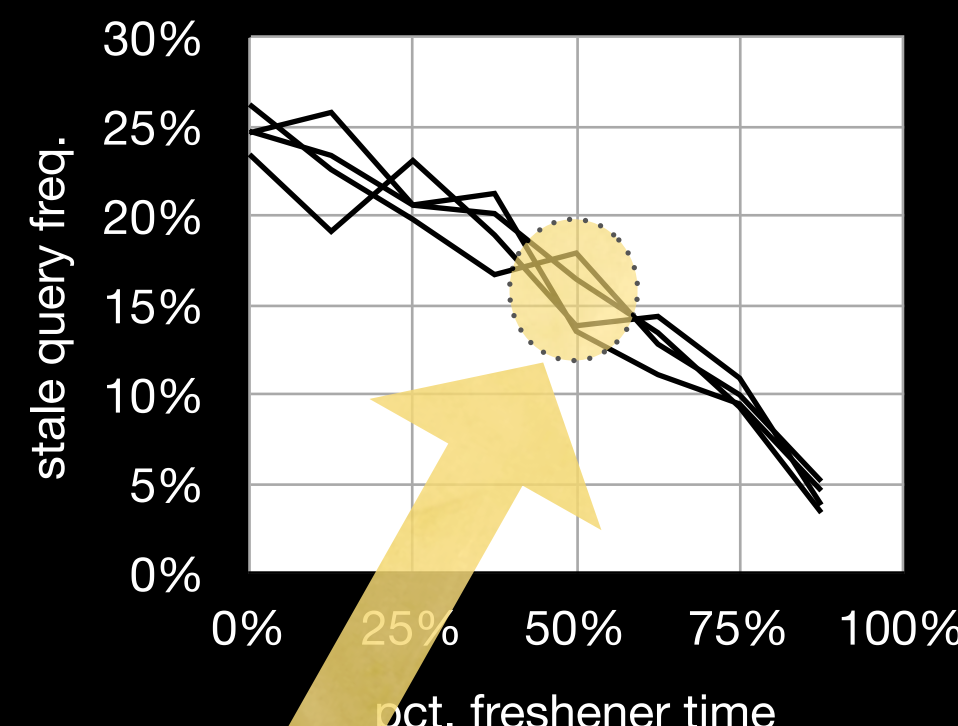
# Invalidation + Breadcrumbs + Round-Robin Fresheners

1 year Fish Market, Smalltalk model

Mac:
always use 8 cores
0 to 7 fresheners
8 down to 1 workers



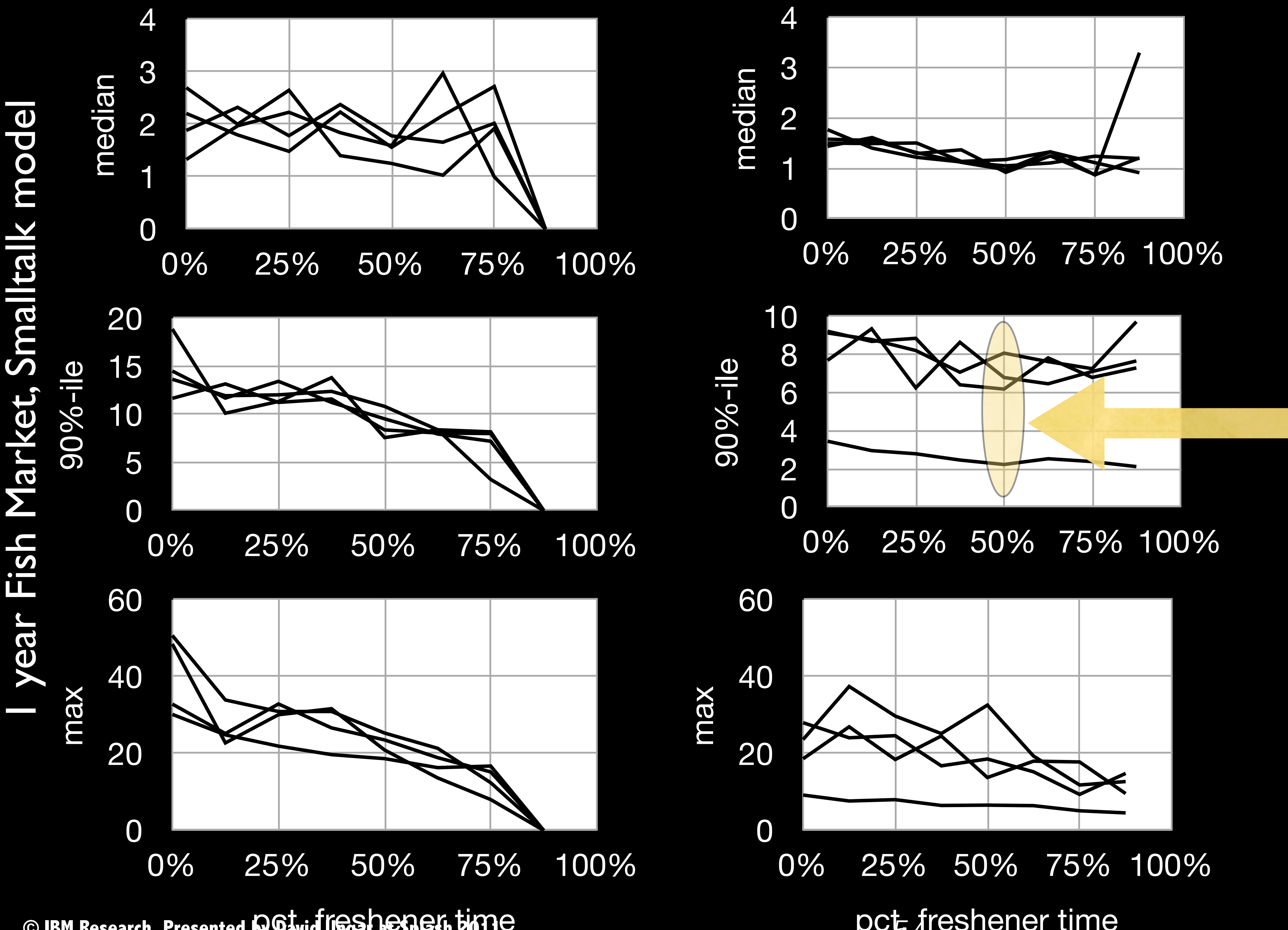

Tilera:
always use 16 cores
0 to 14 fresheners
16 down to 2 workers




Example:
With one freshener per worker, < 20% of results were stale

# How stale?

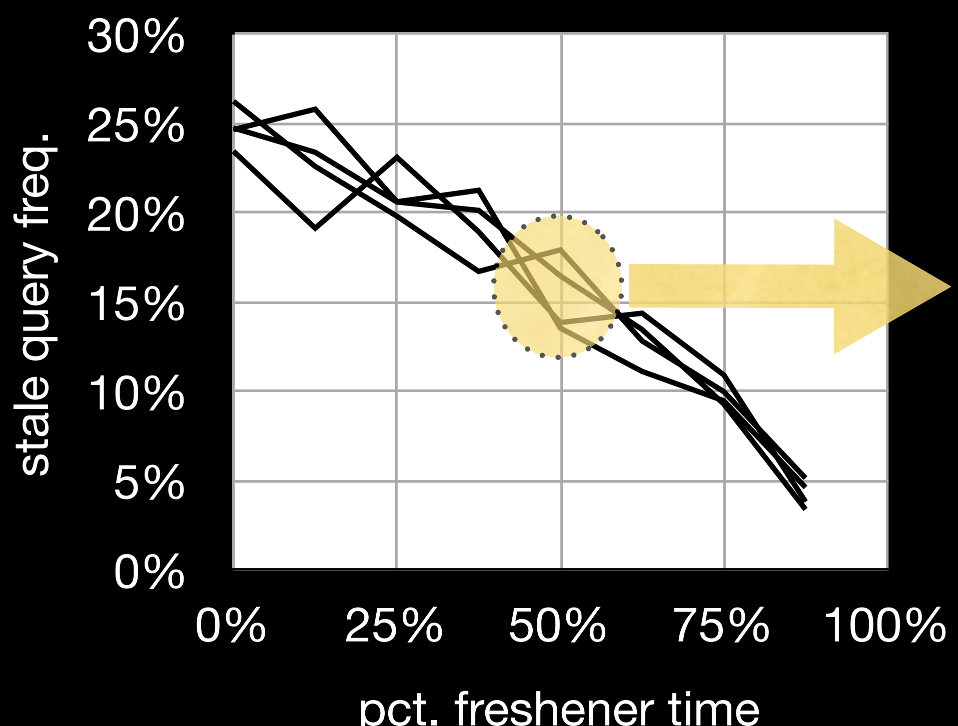
| Mac: | Tilera: |
|---|---|
| always use 8 cores | always use 16 cores |
| 0 to 7 fresheners | 0 to 14 fresheners |
| 8 down to 1 workers | 16 down to 2 workers |




at 1 freshener per 1 worker, 90% of the stale results are < 8 query times stale.

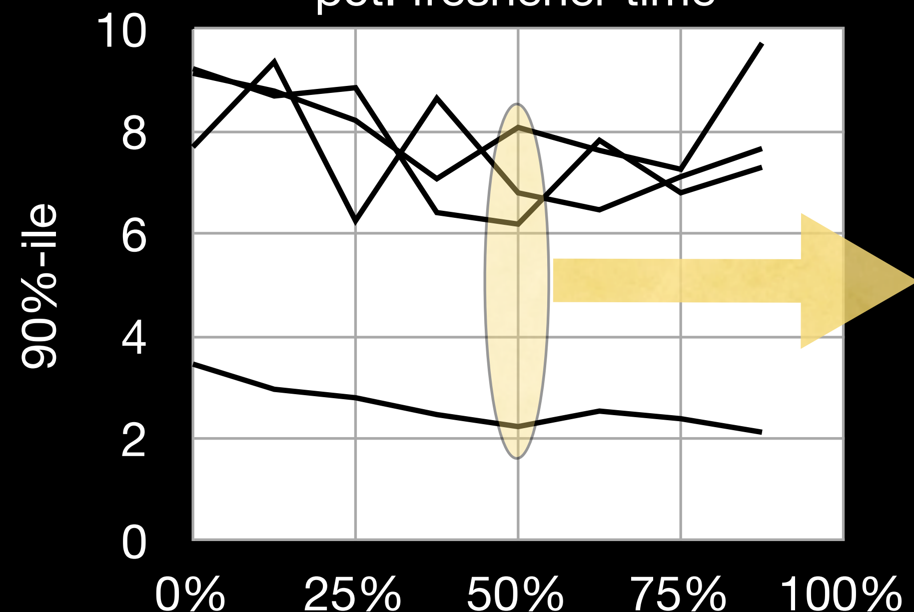
# How often & how stale?

Tilera: using 16 cores, 0 – 14 fresheners, 16 – 2 workers16 down to 2 workers




With one freshener per worker,
< 20% of results were stale

So, only 2% of all queries return results staler than 8.




at 1 freshener per 1 worker,
90% of the stale results are < 8 query times stale

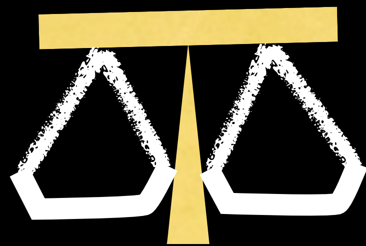
# Summary: Fresheners

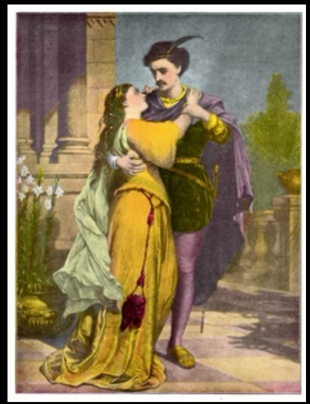
- Instead of synchronizing cache invalidation with recomputation, allow data race errors
  - **Freshen** possibly-stale caches in parallel
- < 2% queries staler than 8 query times
- Race & Repair: Antilock Computing

Embrace and manage inconsistency to enable scaling

Inconsistency Robustness for Scalability in Interactive Concurrent-Update In-Memory MOLAP Cubes,
with Kimelman & Adams

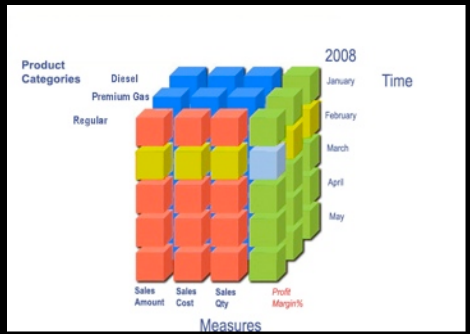
Fundamental





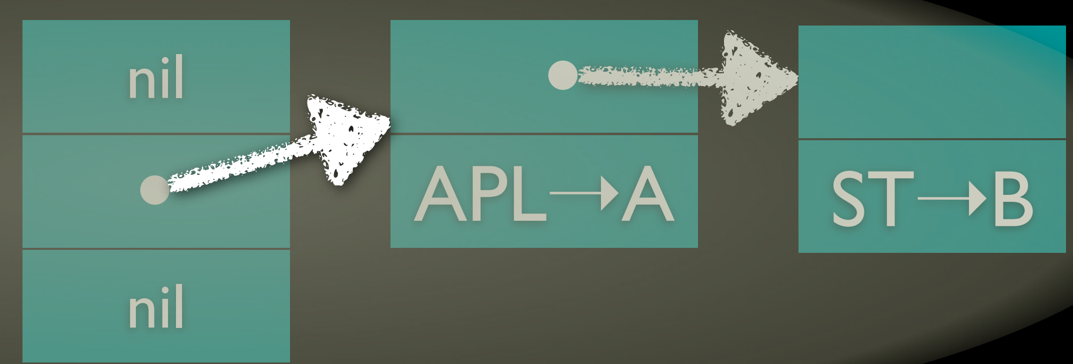
Ensembles & Adverbs

Fresheners & Breadcrumbs




Mitigate, Race, Repair

Locals & Breadcrumbs



# Background



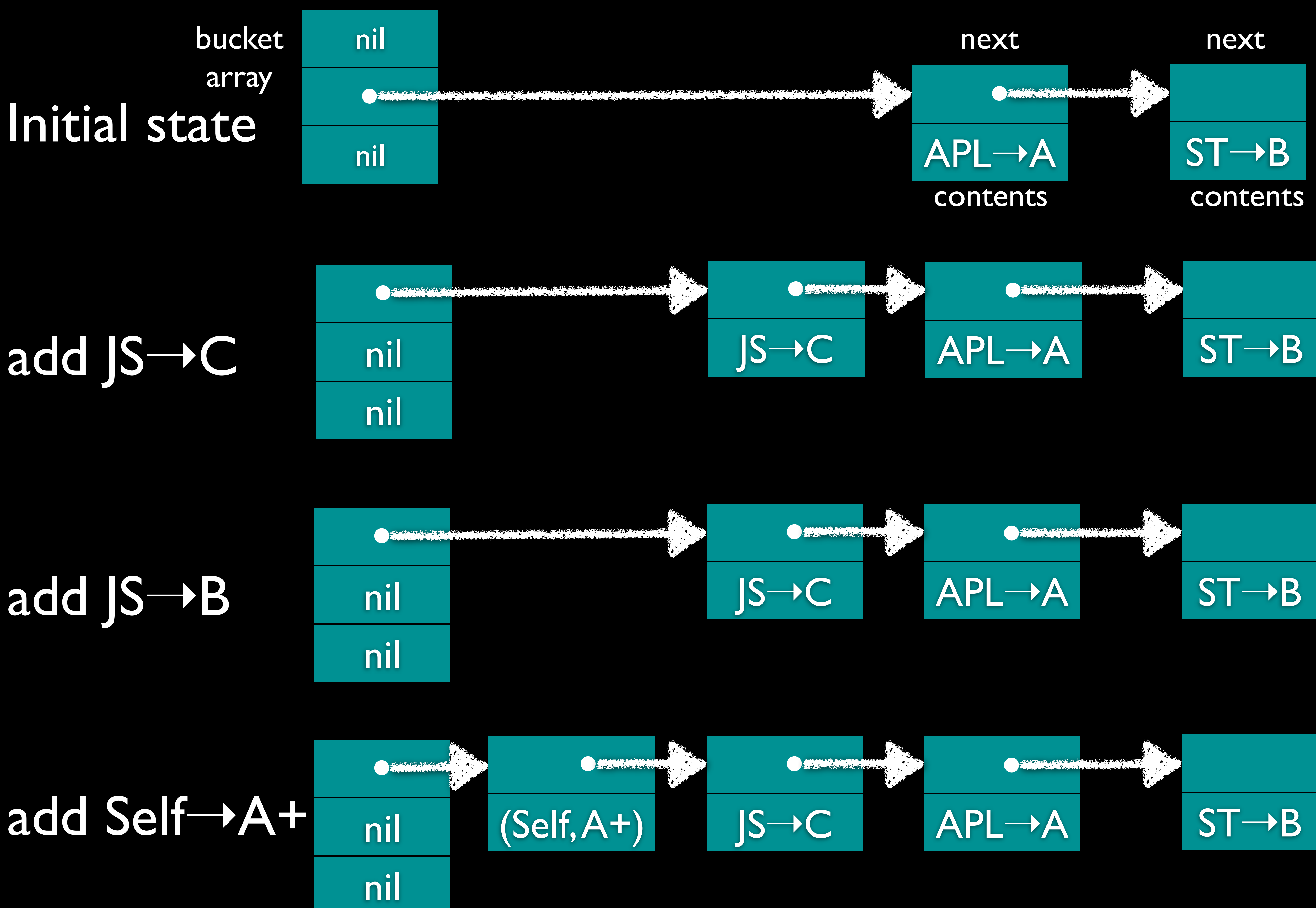




58

# Adding cells to our Cube

- Example: adding a new quarter of fish data
- Cells accessed by hash tables
- What happens without sync?


Initial state
bucket array
nil
nil
next
next
APL→A
ST→B
contents
contents
add JS→C
nil
nil
JS→C
APL→A
ST→B
add JS→B
nil
nil
JS→C
APL→A
ST→B
add Self→A+
nil
nil
(Self, A+)
JS→C
APL→A
ST→B

# add(assoc)

```
for ( node = buckets[assoc->key->hash()];
        node != NULL;
        node = node->next)
    if (node->contents->key == assoc->key)
         return // already there!
new_node = new Node()
new_node->contents = assoc
new_node->next = buckets[assoc->key->hash()]
buckets[assoc->key->hash()] = new_node
```

# add(assoc)

```
for ( node = buckets[assoc->key->hash()];
      node != NULL;
      node = node->next)
    if (node->contents->key == assoc->key)
      return // already there!
new_node = new Node()
new_node->contents = assoc
new_node->next = buckets[assoc->key->hash()]
buckets[assoc->key->hash()] = new_node
```







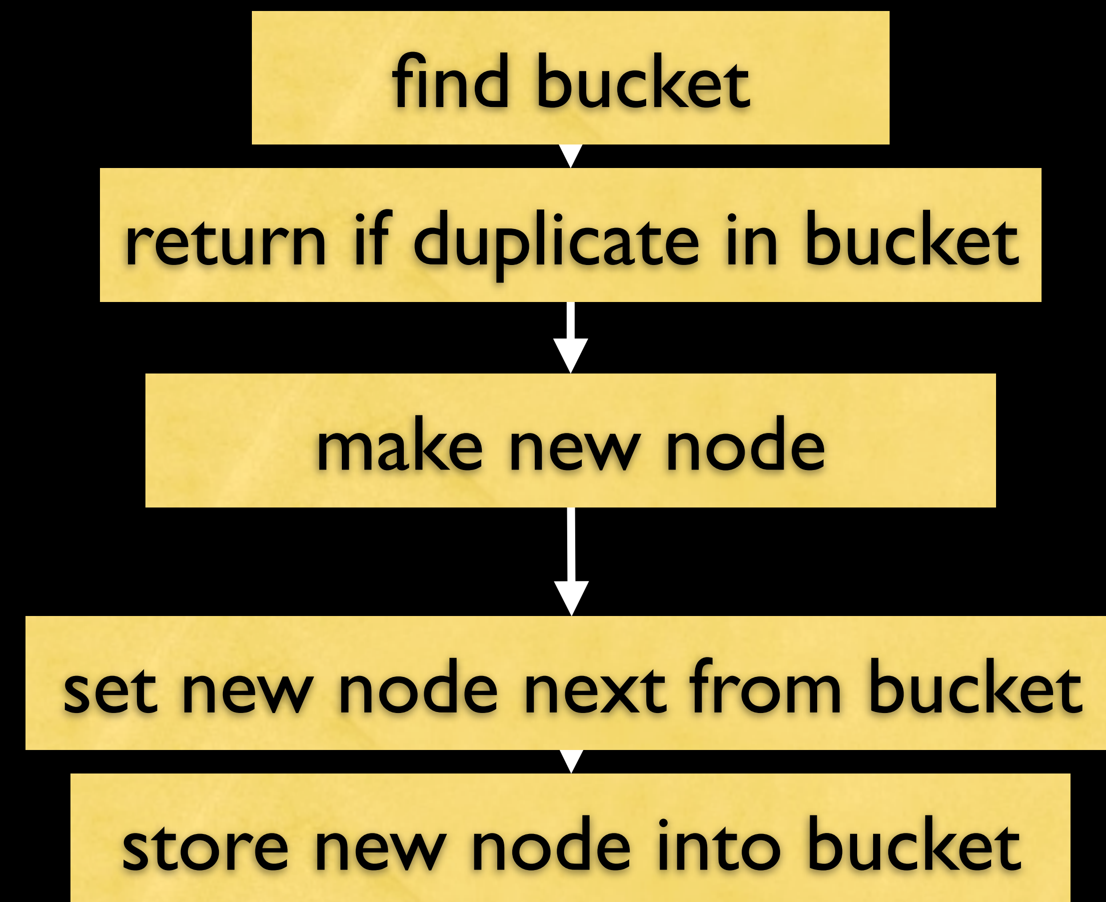
62

# add(assoc)

```
bp = &buckets[assoc->key->hash()]
for ( node = *bp;
      node != NULL;
      node = node->next)
    if (node->contents->key == assoc->key)
        return // already there!
new_node = new Node()
new_node->contents = assoc
new_node->next = *bp
*bp = new_node
```

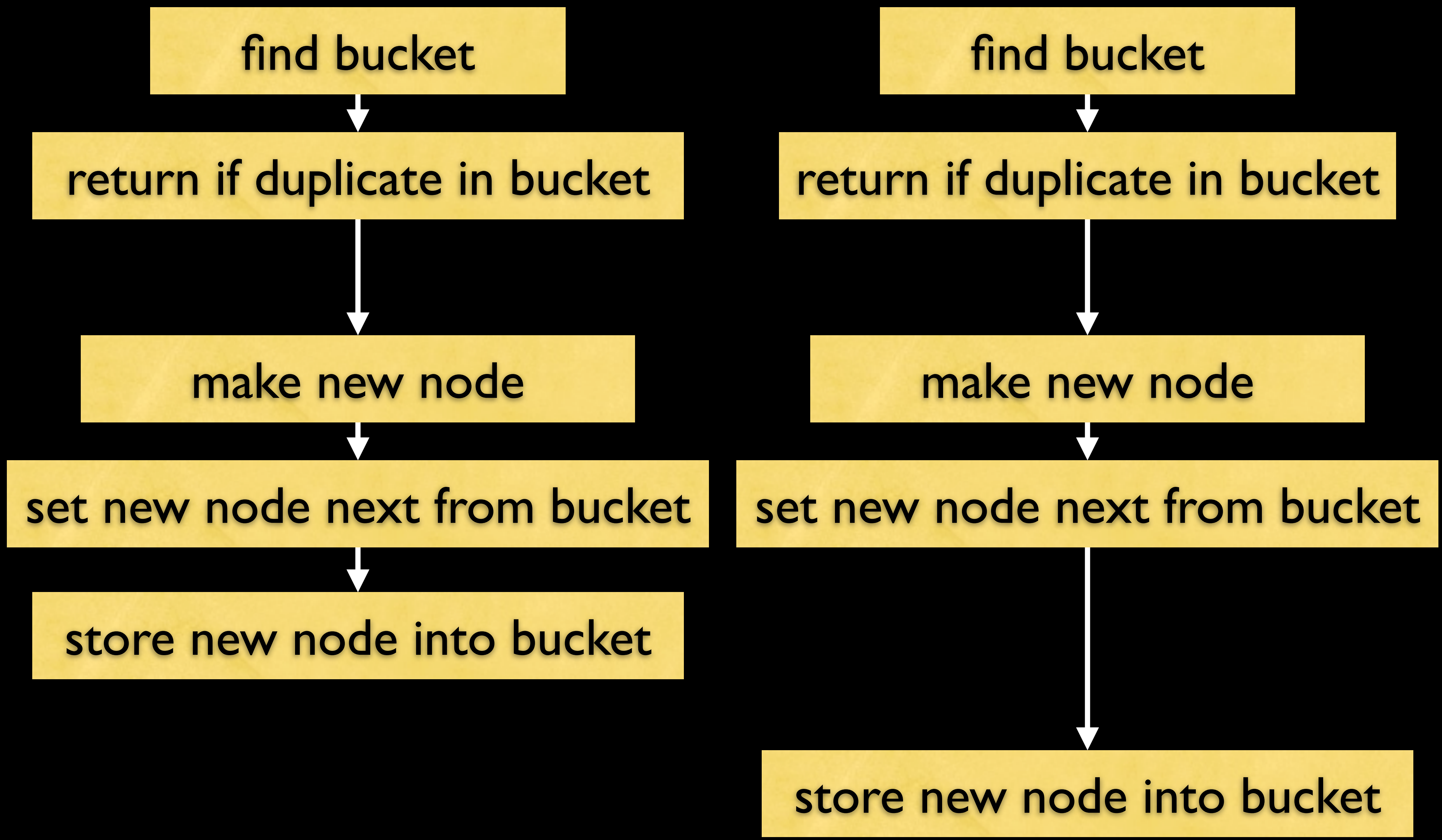
# add(assoc)

```
bp = &buckets[assoc->key->hash()]
<return if duplicate at *bp>
new_node = new Node()
new_node->contents = assoc
new_node->next = *bp
*bp = new_node
```

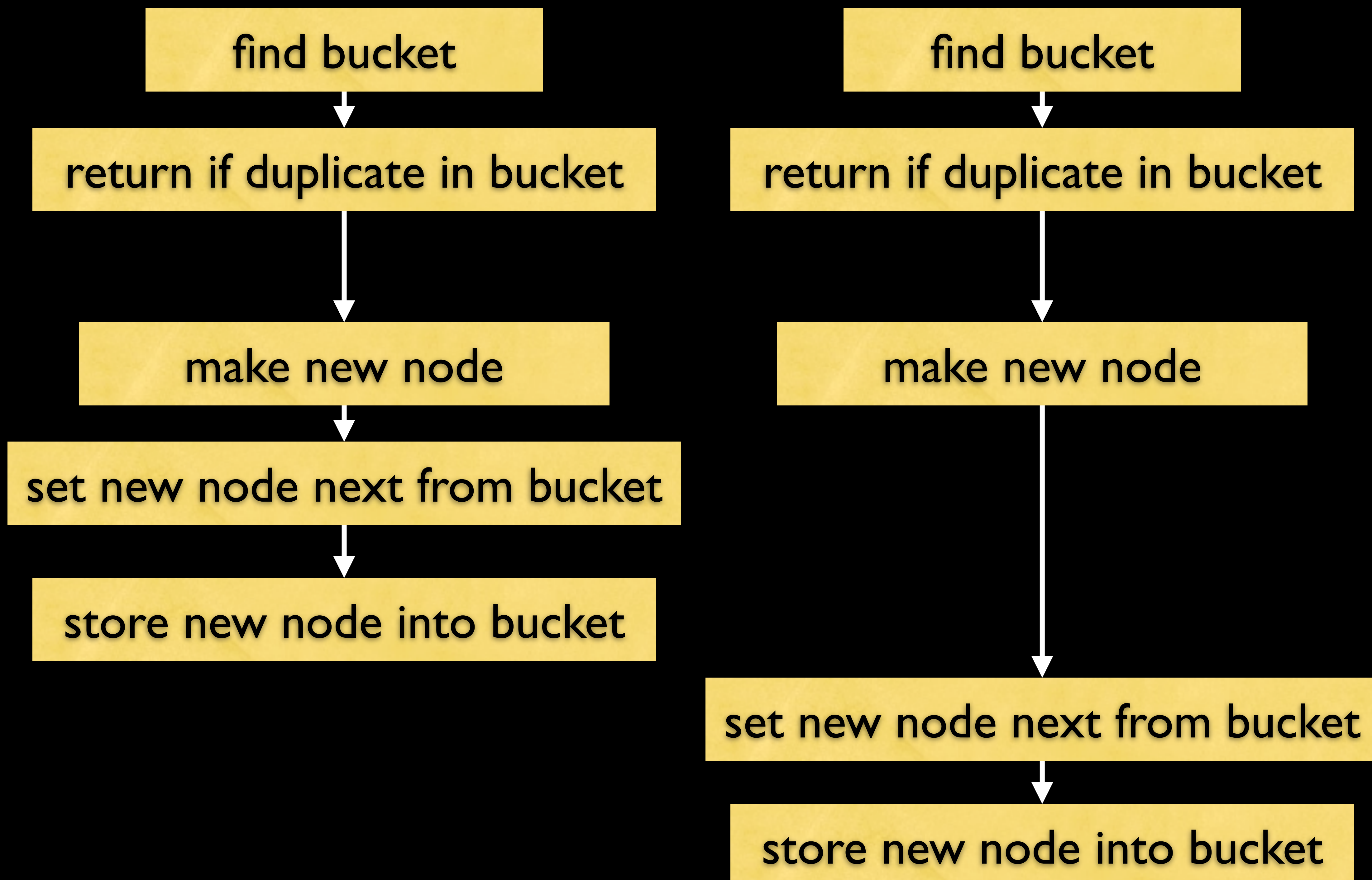


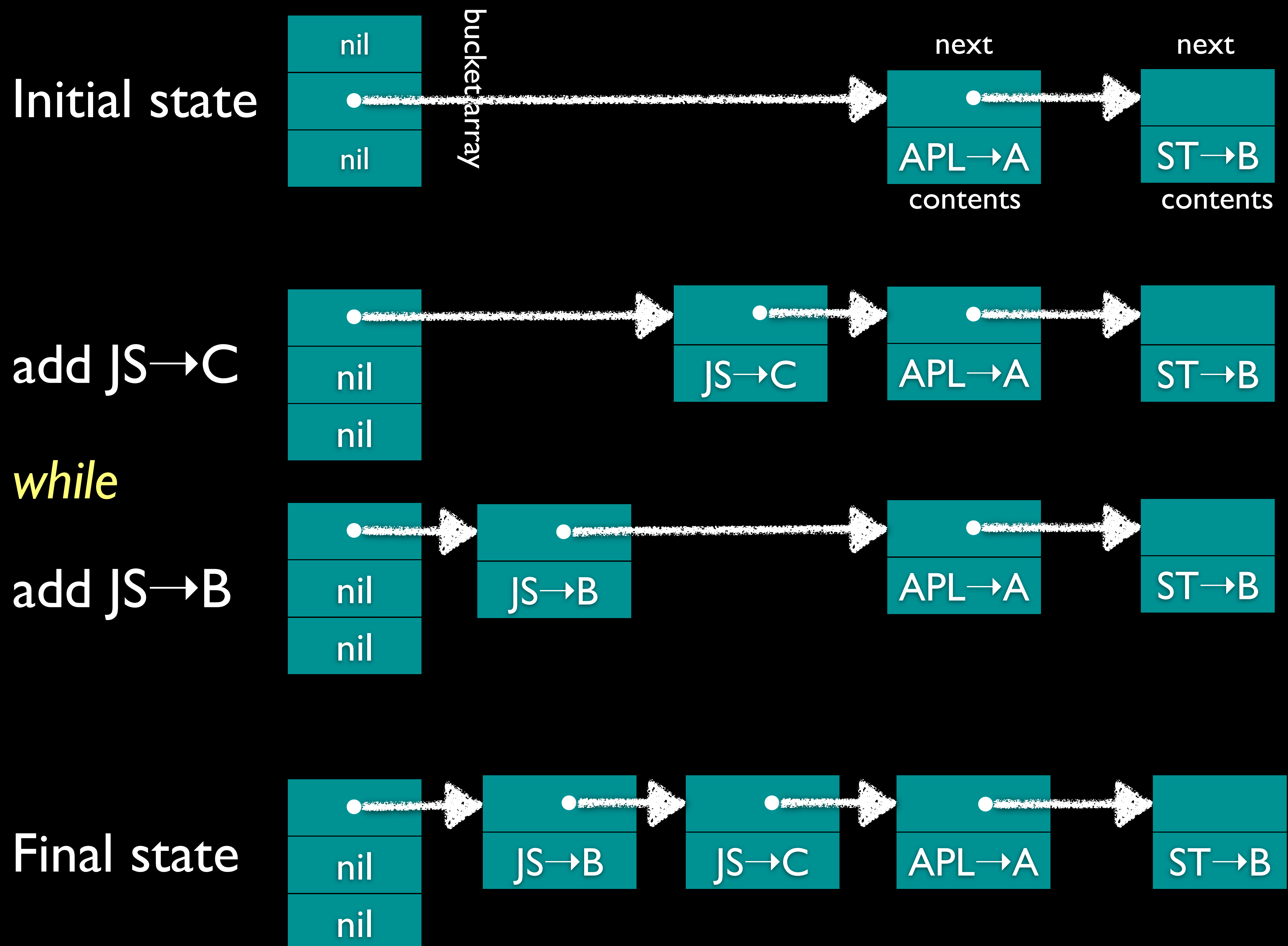
# Parallel Chaos

# Interleavings: one winner
# Can miss an insertion



# Interleavings: two winners: can add same key twice




Initial state
nil
bucket array
nil
next
next
APL→A
ST→B
contents
contents
add JS→C
nil
nil
JS→C
APL→A
ST→B
while
add JS→B
nil
nil
JS→B
APL→A
ST→B
Final state
nil
nil
JS→B
JS→C
APL→A
ST→B

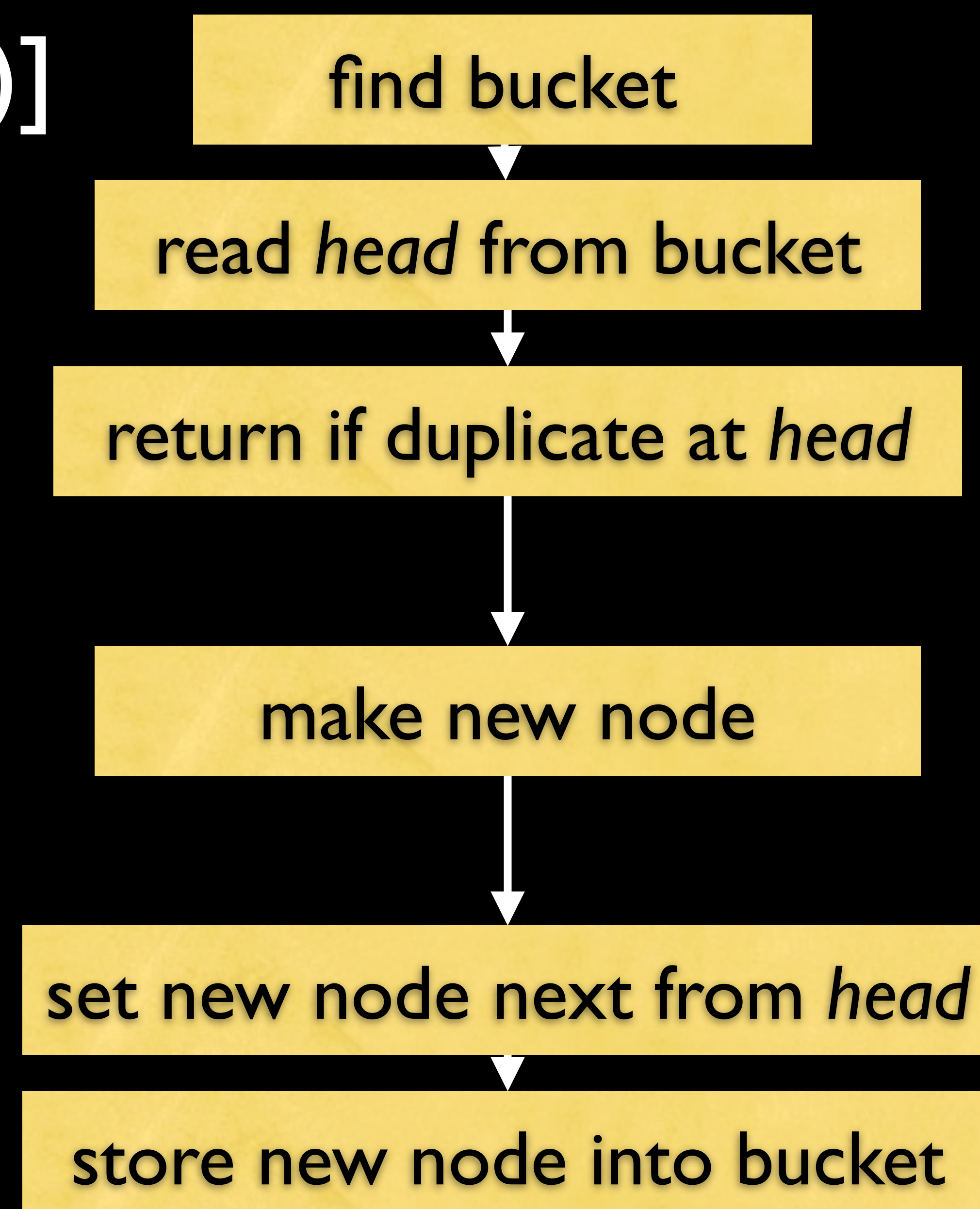
# Bounding the error

# A simple fix, without synchronization

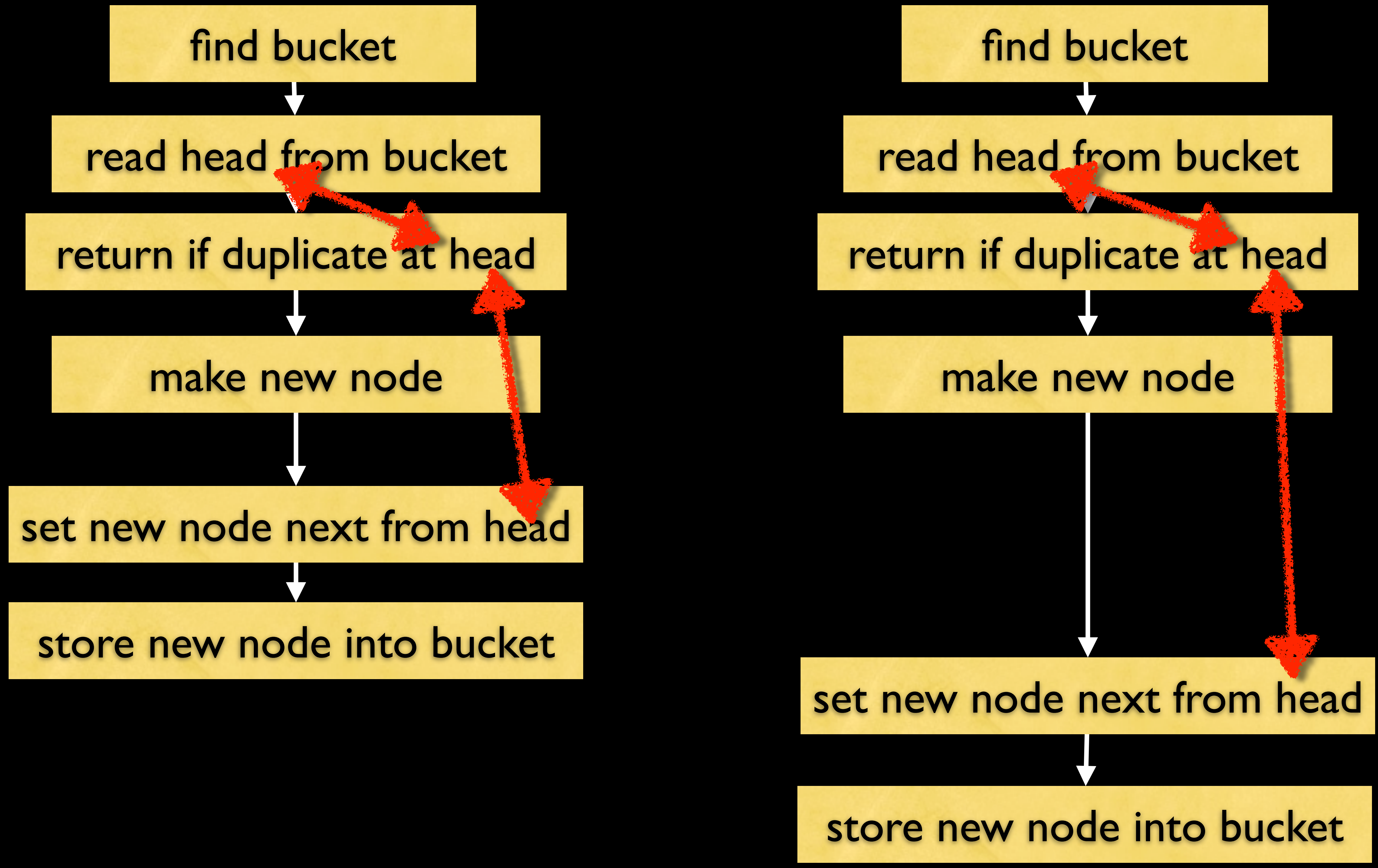
```
bp = &buckets[assoc->key->hash()]
head = *bp
for ( node = head;  node != NULL;
       node = node->next)
    if (node->contents->key == assoc->key)
        return; // already there!
new_node = new Node();
new_node->contents = assoc;
new_node->next = head
*bp = new_node
```

# add(assoc)

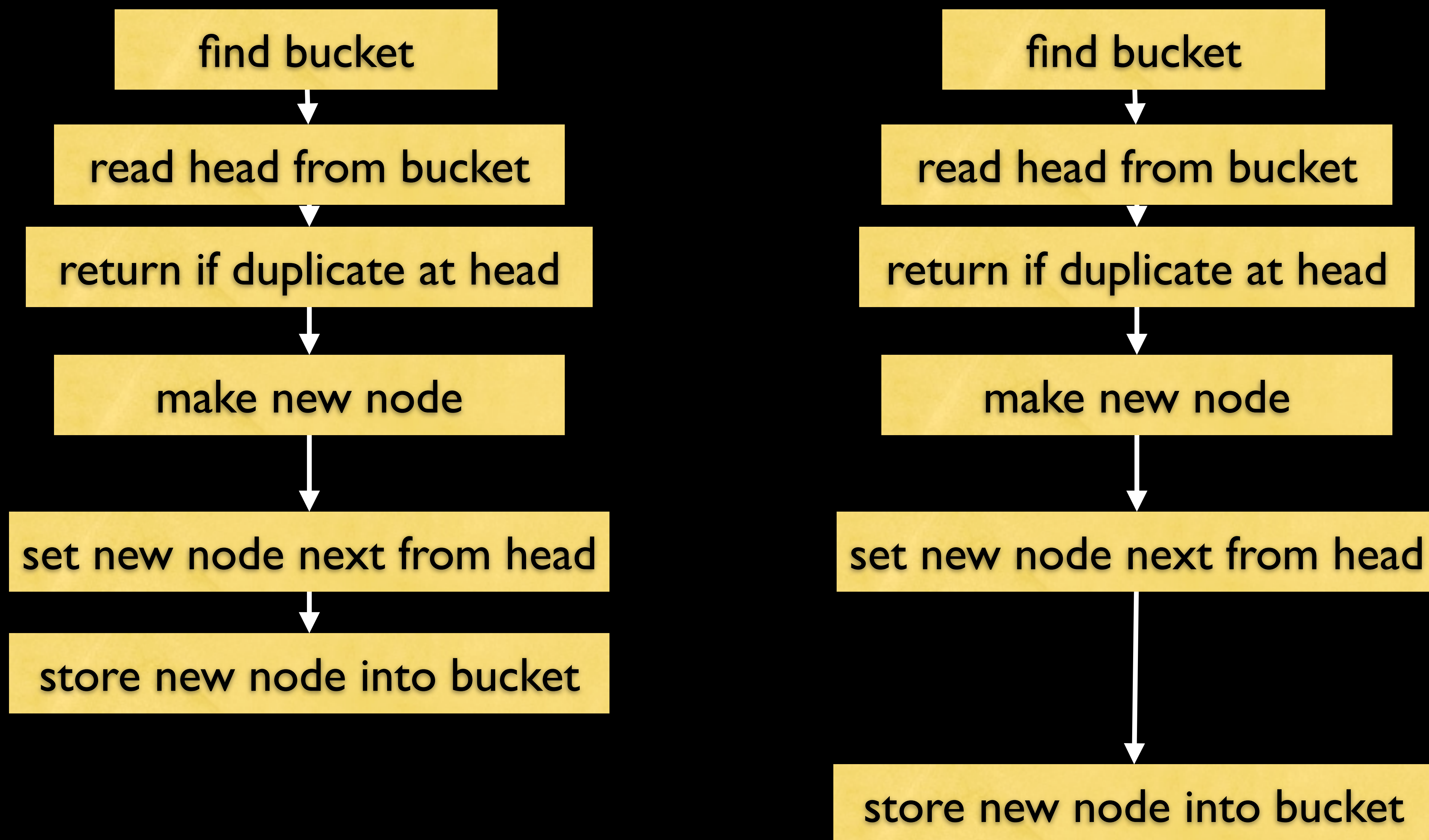
```
bp = &buckets[assoc->key->hash()]
head = *bp
<return if duplicate at head>
new_node = new Node()
new_node->contents = assoc
new_node->next = head
*bp = new_node
```



# Cannot add same key twice! Despite unsynchronized



# Can still fail to insert different key

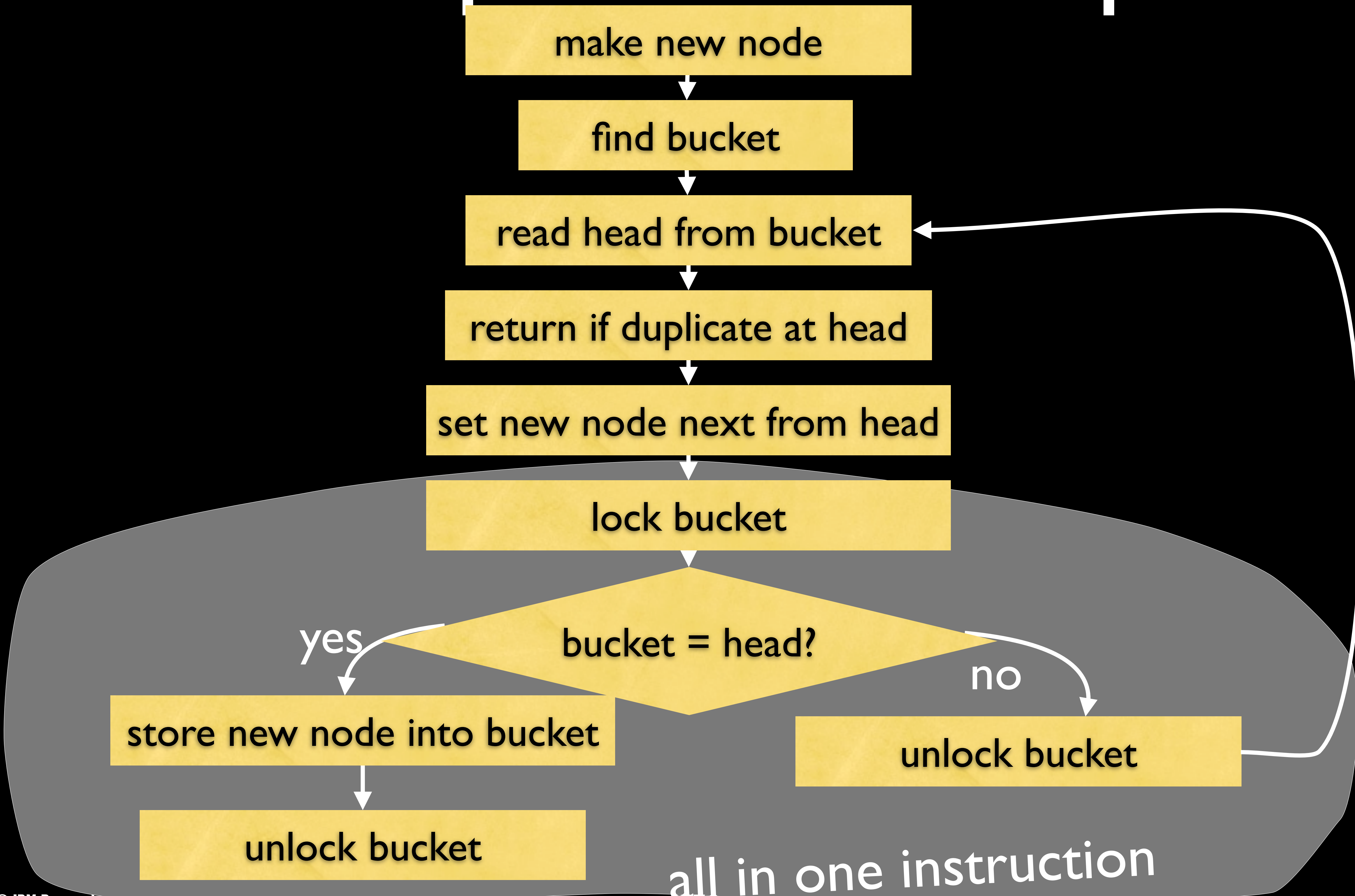


# Mitigation Strategies

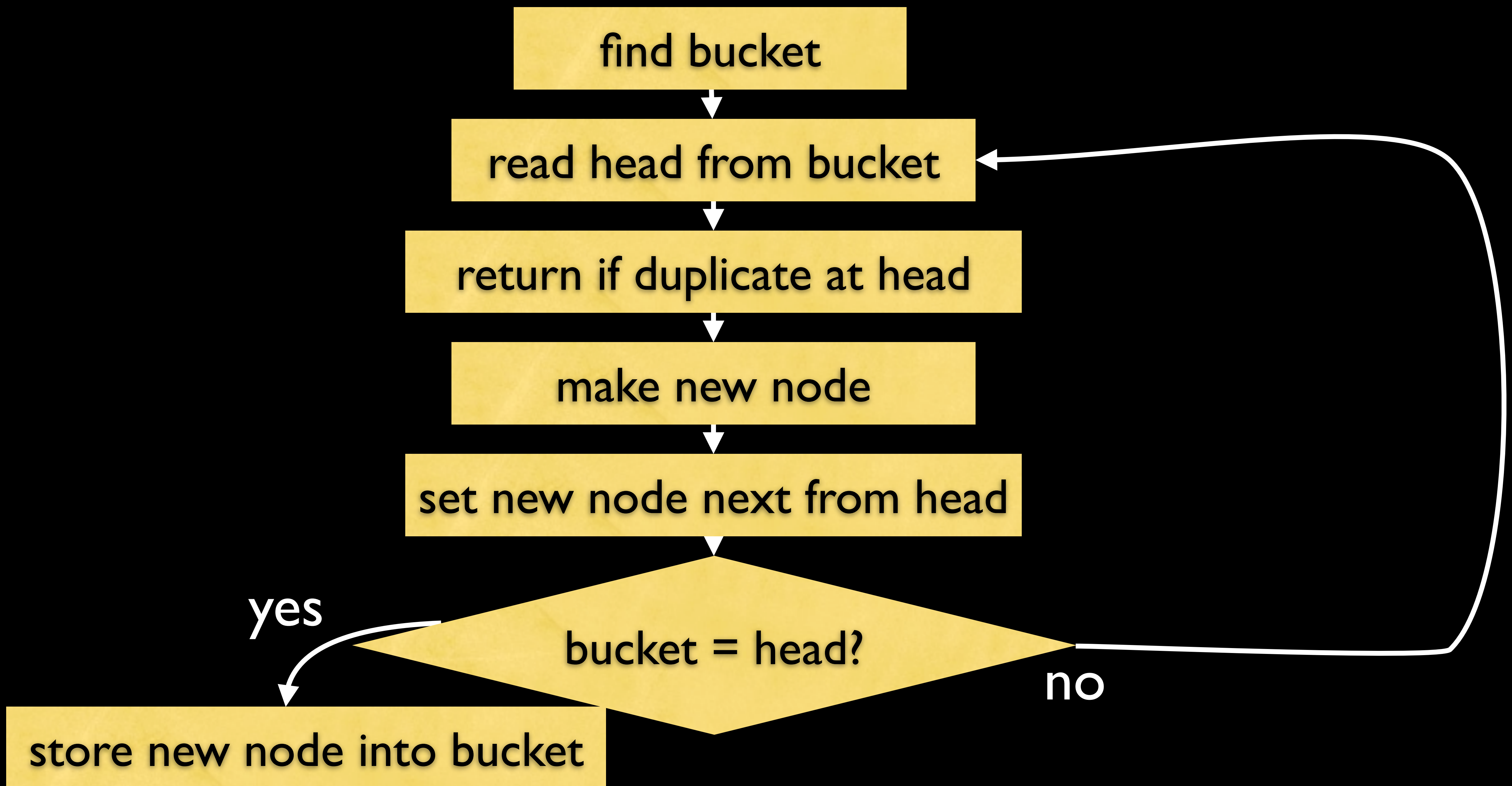
# No check

make new node
↓
find bucket
↓
read head from bucket
↓
return if duplicate at head
↓
set new node next from head
↓
store new node into

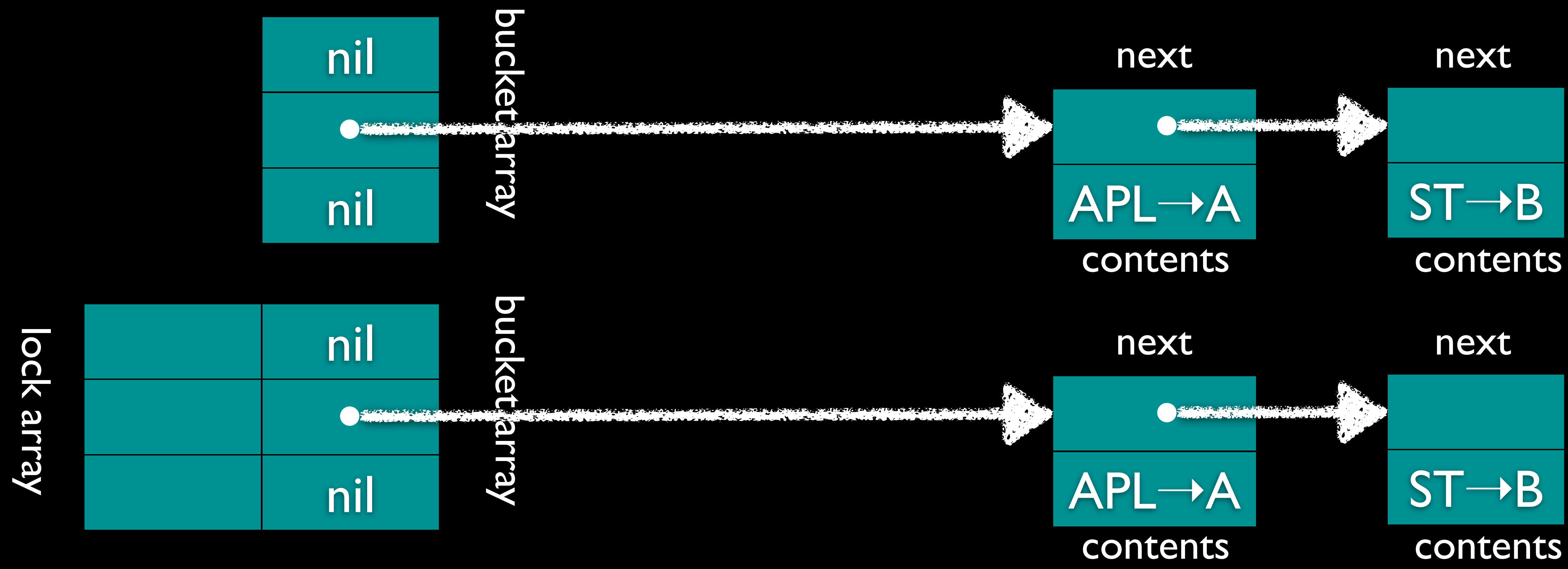
# Compare-and-Swap



# Check head before store

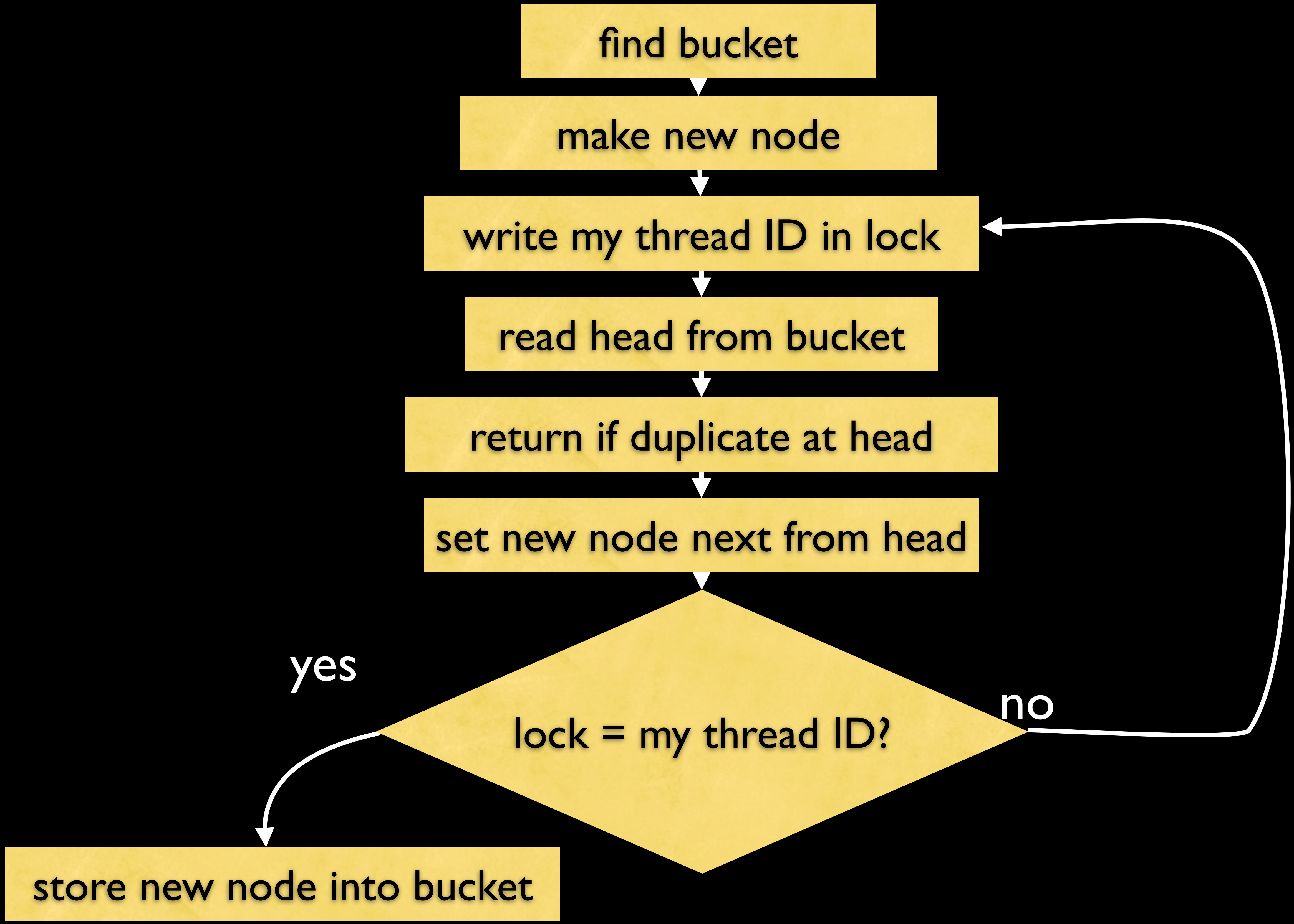


# Intention locks

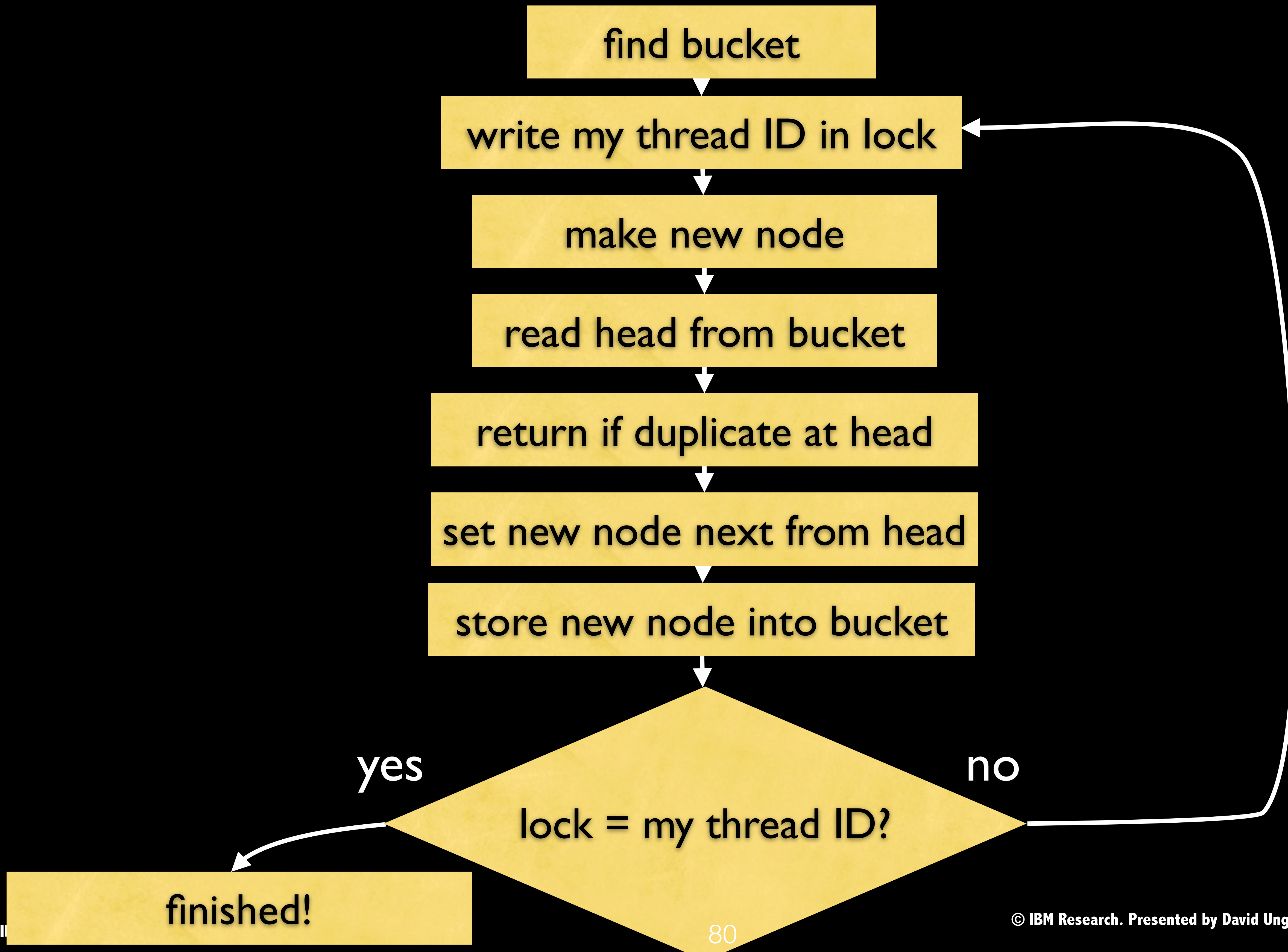



Put thread ID in lock when starting,
Check lock before/after store

# Intention lock check before



# Intention lock check after



# Mitigation Strategies

- Atomic instruction for storing head (lock-free approach)
- Check bucket before storing head
- Check intention-lock before and/or after store
- Just pass the buck to a higher level

## Which would you choose?

# The Experiment

# Experiments

- Platform: 8-core Mac
  - Multicore, not manycore
- Varying # threads: 1, 8
- Varying list strategies
- Varying experiments

# List strategies

- unchecked
- check list head
- check intention lock
  - before
  - after
  - before & after
- compare-and-swap (CAS)

# Experiments

- no contention: each thread inserts into a different list
- max contention: each thread inserts into the same list
- max with retries: after each insert attempt:
  - wait insert time, exit if insert succeeded
  - if not, binary exponential backoff (<128)

# Results





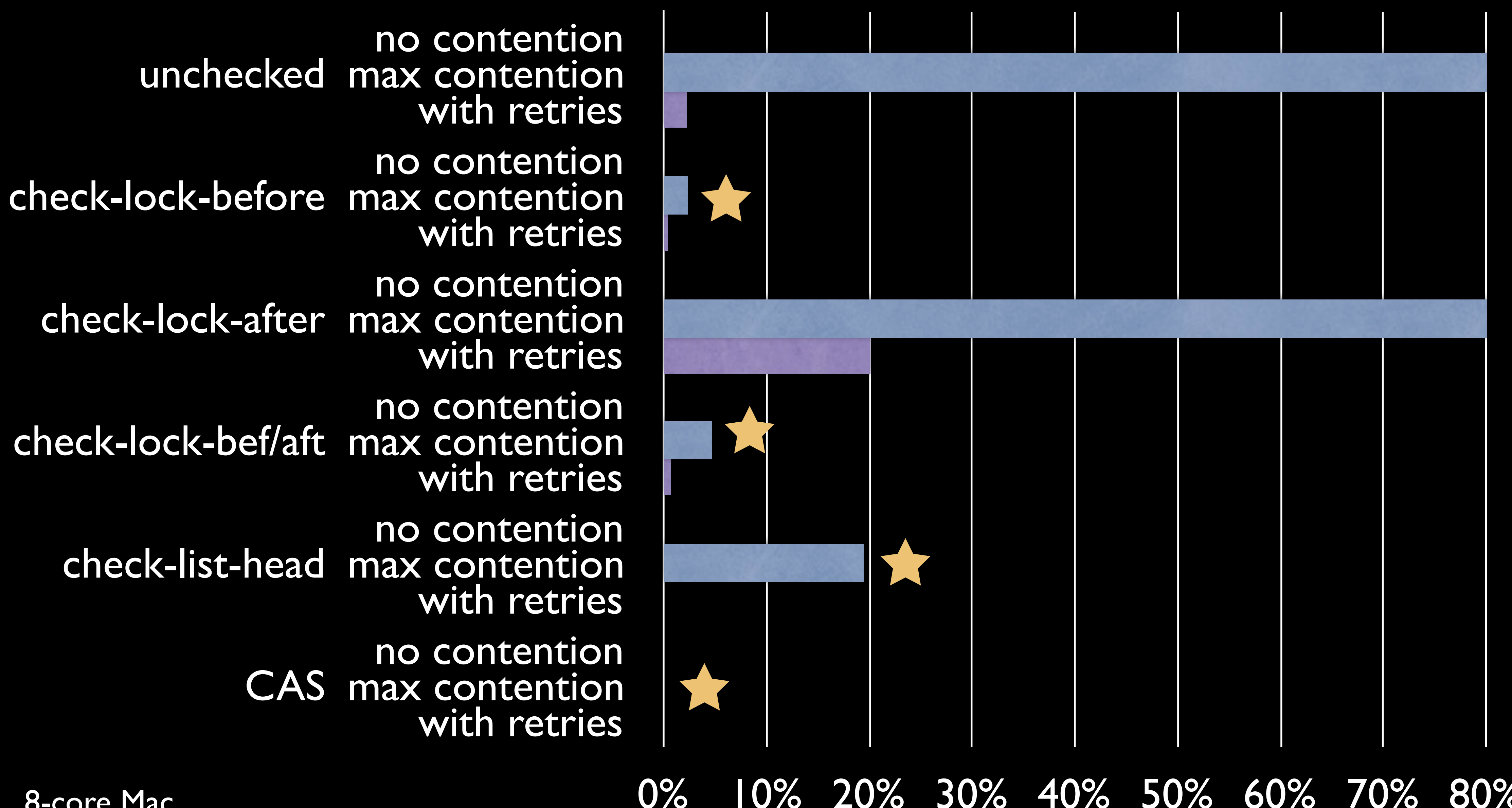


86

# Disclaimer:
# Unreviewed Work!!!
# Contains errors

# Miss rate results

- Miss rate: how many insertion attempts fail
  - no contention: no misses




8-core Mac

# Iterations: How many times around the loop? (mean iterations per insert attempt)

8-core Mac Pro

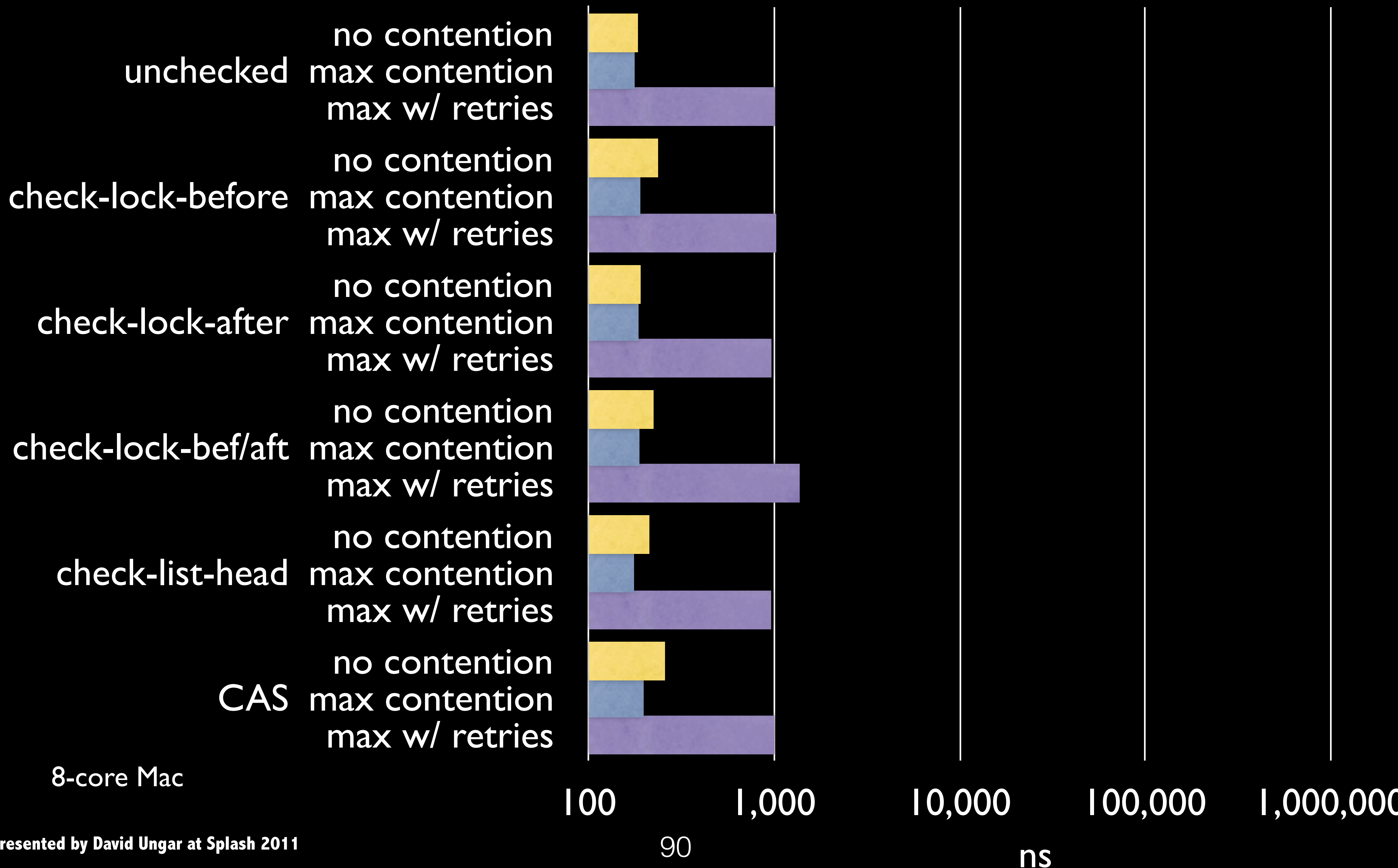




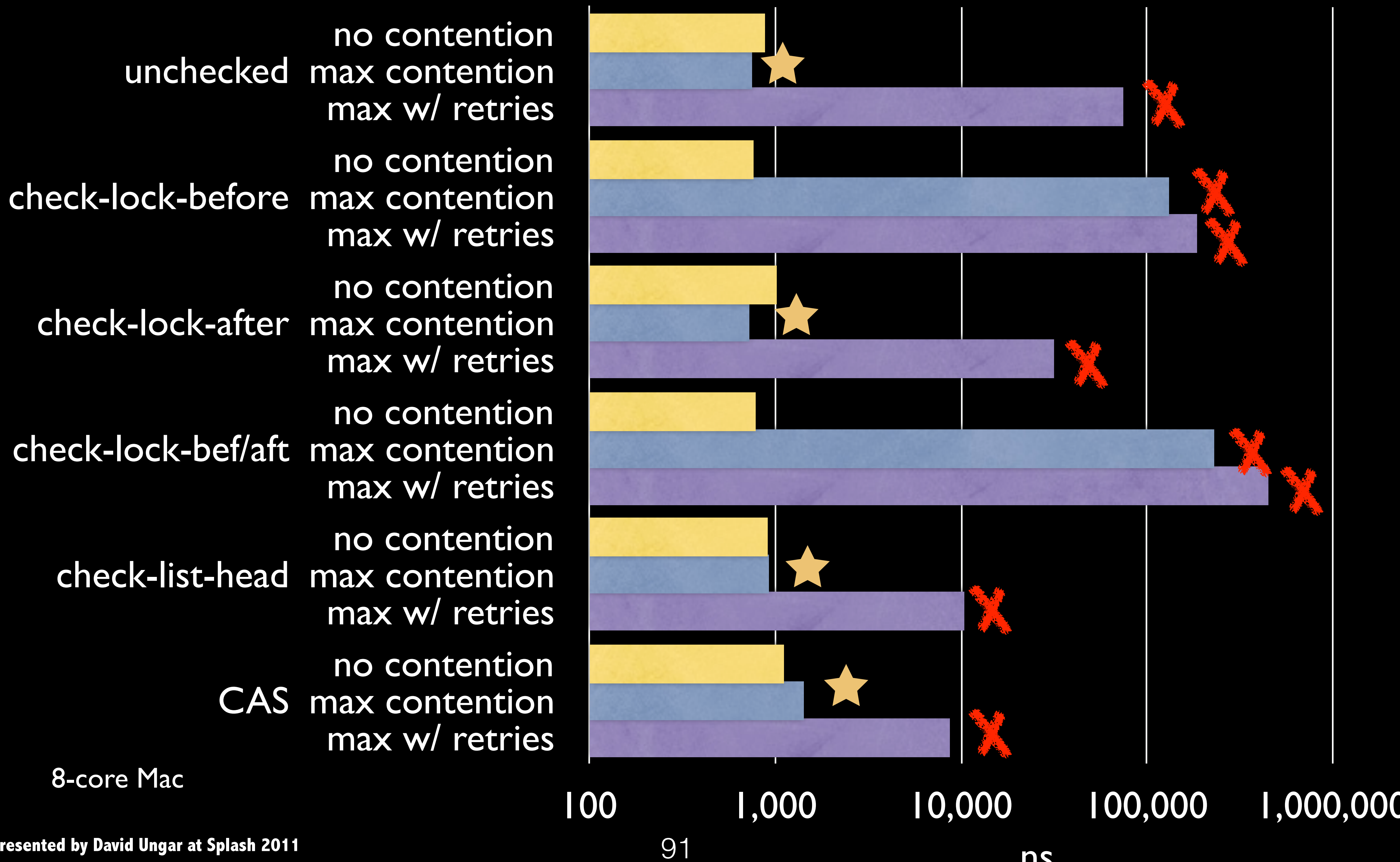



89

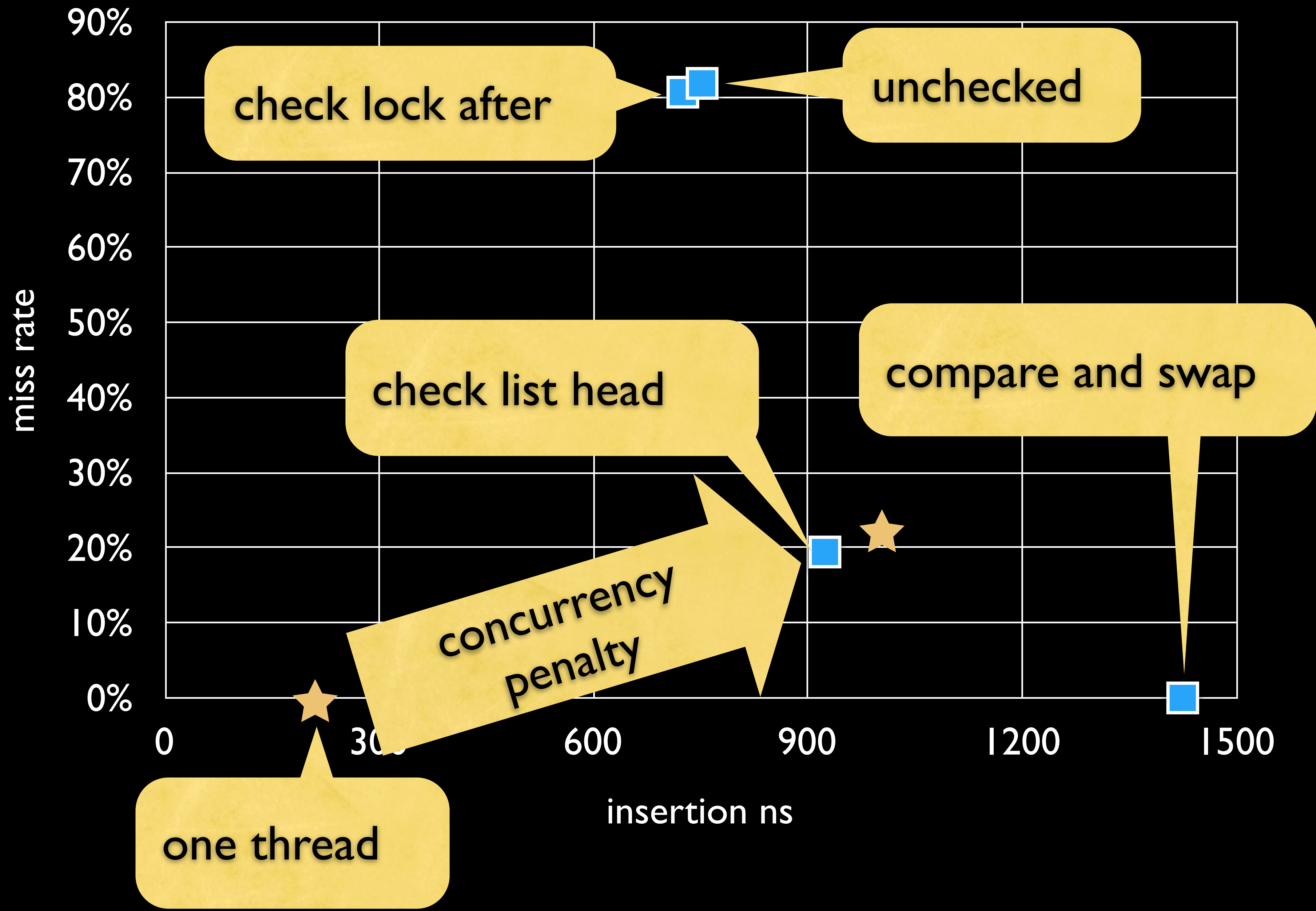
# How much time per insert attempt? (Excluding duplicate-search time) one thread



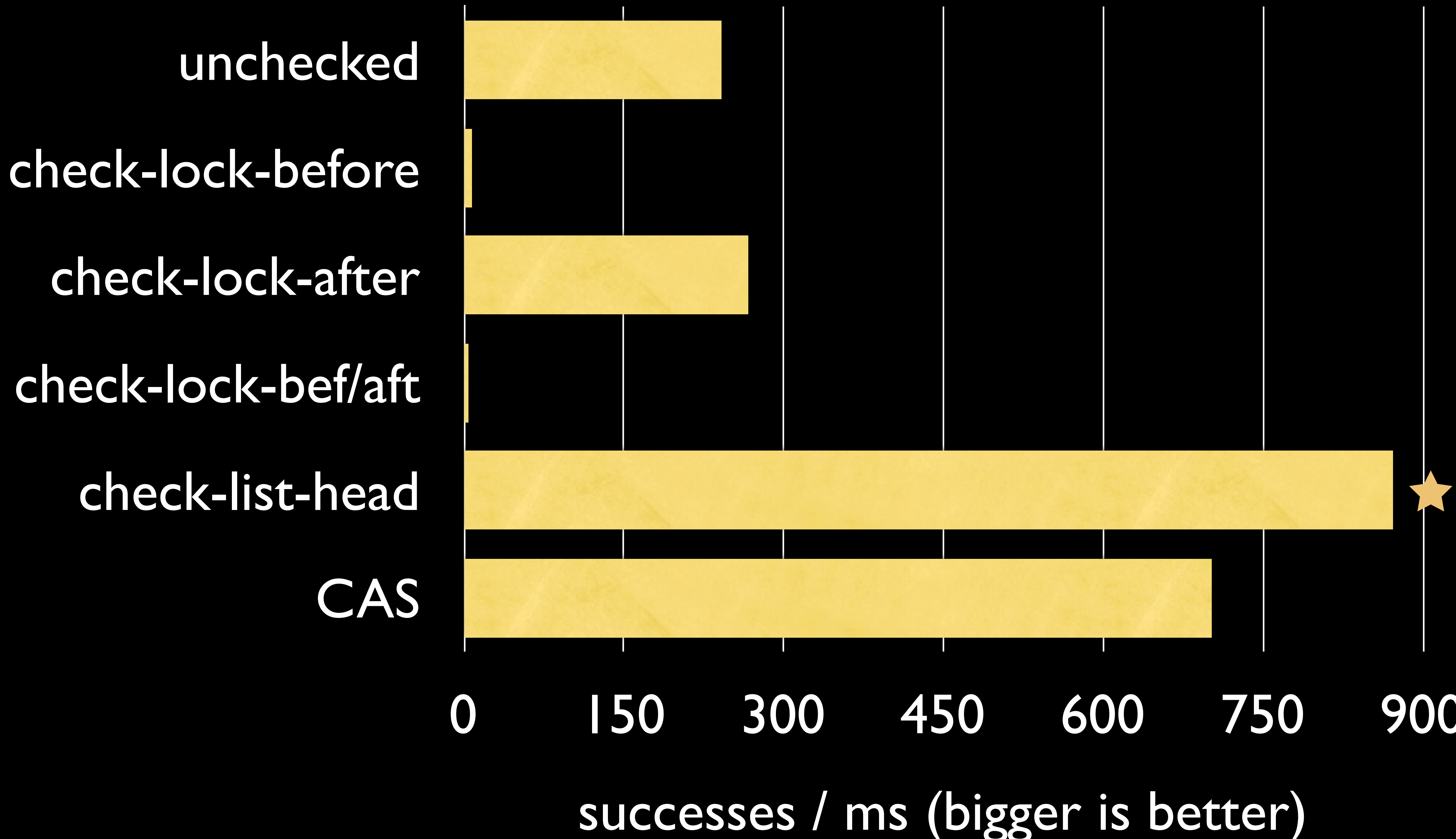
# How much time per insert attempt? (Excluding duplicate-search time) 8 threads



# Miss rate vs time, 8-core Mac



$$\frac{successes}{ms} = \frac{successes}{attempt} \times \frac{attempts}{ms}$$




(But **one** thread = 5,600)

8-core Mac

# Summary: Parallel Sets

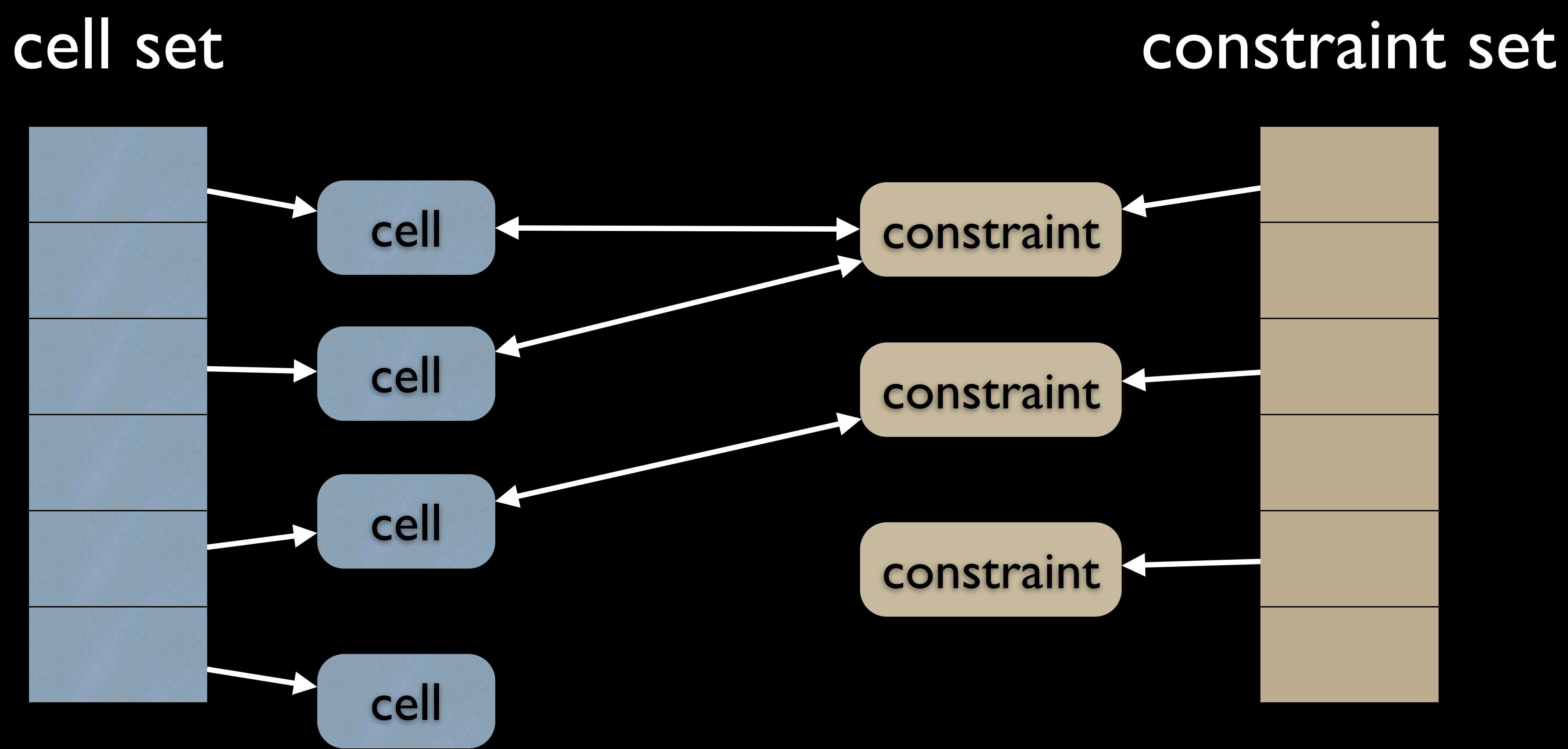
- Probabilistic data structures:
  - New area?
- Hypothesis: accuracy trades off against performance
  - CAS may not win
  - Big penalty on current hardware





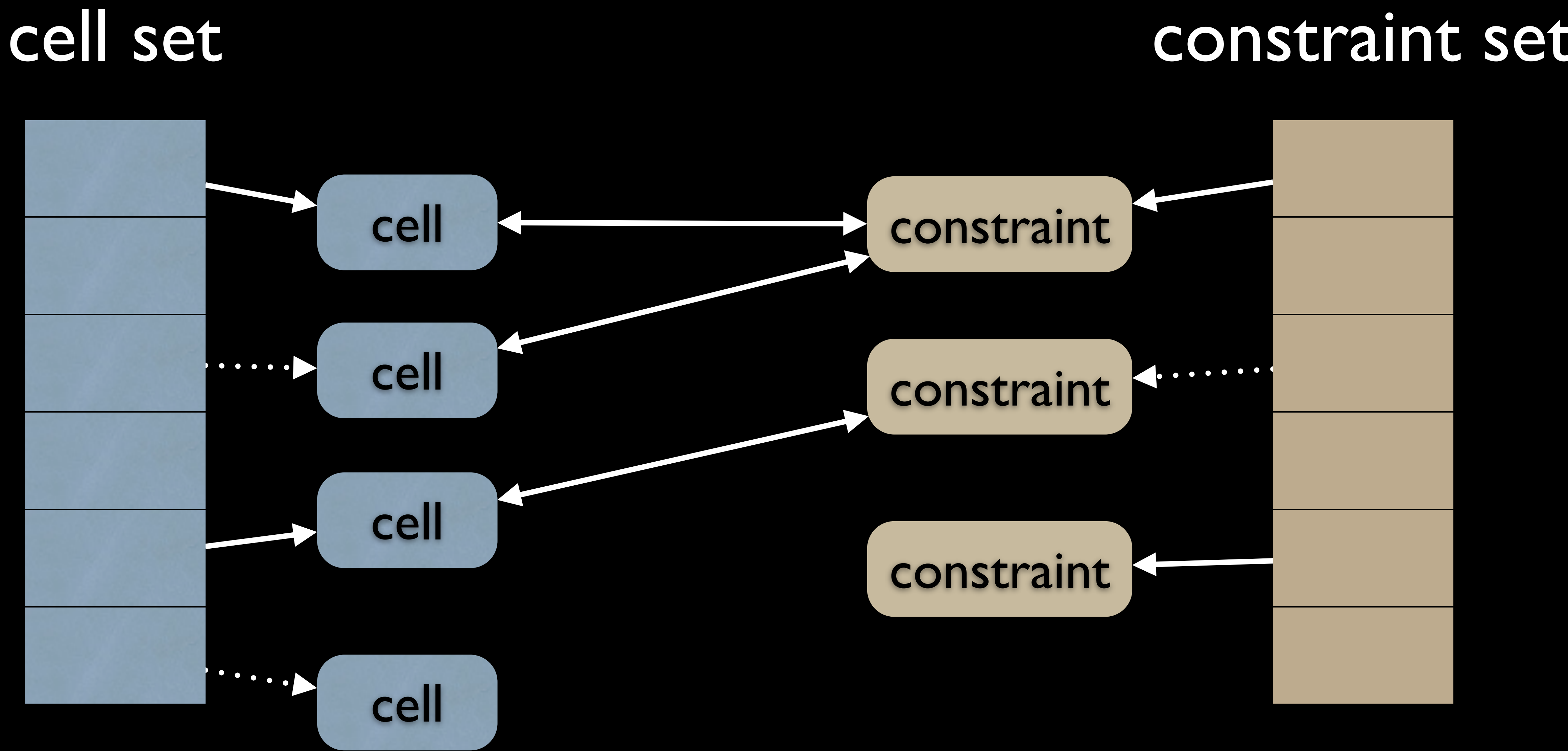
94

# An aside: freeing




Easy if you can count on the invariants

# An aside: freeing




Harder if you cannot count on the invariants

# Conclusion

- Hardware trends will force us to give up on certainty, determinism, repeatability
- Good enough, soon enough, race-and-repair, anti-lock
- A different way of thinking
  - invariants become probable
- New data structures & algorithms
- Can we do it?

# Acknowledgements


- IBM partners
  - Sam Adams, Brent Hailpern, Doug Kimelman, Mark Wegman
- Academic partners
  - Andrew Black, Stefan Marr, Theo D'Hondt
- Inspiration
  - Paul McKenney, Jonathan Walpole, Martin Rinard
- Talk help
  - Misha Dmitriev, Peter Kessler, David Leibs, Randy Smith, Michael Vandervanter, Mario Wolczko